NASA Technical Memorandum 108795

IN-61
203569
61P

437364

The Automated Instrumentation and Monitoring System (AIMS) Reference Manual

Jerry Yan, Philip Hontalas, Sherry Listgarten, et al.

(NASA-TM-108795) THE AUTOMATED INSTRUMENTATION AND MONITORING SYSTEM (AIMS) REFERENCE MANUAL (NASA) 61 p

N94-23510

Unclas

G3/61 0203569

November 1993

NASA

National Aeronautics and Space Administration

NASA Technical Memorandum 108795

The Automated Instrumentation and Monitoring System (AIMS) Reference Manual

Jerry Yan, Recom Technologies, San Jose, California
Philip Hontalas, Ames Research Center, Moffett Field, California
Sherry Listgarten, Recom Technologies, San Jose, California
et al.

November 1993

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

LIST OF AUTHORS

Jerry Yan, Recom Technologies, San Jose, California

Philip Hontalas, Ames Research Center, Moffett Field, California

Sherry Listgarten, Recom Technologies, San Jose, California

Charles Fineman, Sterling Software, Palo Alto, California

Melisa Schmidt, Pankaj Mehra, and Sekhar Sarukkai,
Recom Technologies, San Jose, California

Cathy Schulbach, Ames Research Center, Moffett Field, California

Table of Contents

Figures

PRECEDING PAGE BLANK NOT FILMED

PAGE VIII INTENTIONALLY BLANK

Summary

Whether a researcher is designing the "next parallel programming paradigm", another "scalable multiprocessor" or investigating resource allocation algorithms for multiprocessors, a facility that enables parallel program execution to be captured and displayed is invaluable. Careful analysis of execution traces can help computer designers and software architects to uncover system behavior and to take advantage of specific application characteristics and hardware features. A software tool kit that facilitates performance evaluation of parallel applications on multiprocessors is described in this paper. The Automated Instrumentation and Monitoring System (AIMS) has four major software components: a source code instrumentor which automatically inserts active event recorders into the program's source code before compilation; a run-time performance-monitoring library, which collects performance data; a trace file animation and analysis tool kit which reconstructs program execution from the trace file; and a trace post-processor which compensate for data collection overhead. Besides being used as a prototype for developing new techniques for instrumenting, monitoring, and visualizing parallel program execution, AIMS is also being incorporated into the run-time environments of various hardware testbeds to evaluate their impact on user productivity. Currently, AIMS instrumentors accepts FORTRAN and C parallel programs written for Intel's NX operating system on the iPSC family of multicomputers. A run-time performance-monitoring library for the iPSC/860 is included in this release. We plan to release monitors for other platforms (such as PVM and TMC's CM-5) in the near future. Performance data collected can be graphically displayed on workstations (e.g., Sun Sparc and SGI) supporting X-Windows (in particular, *X11R5*, `Motif 1.1.3`).

1. Introduction

1.1. The Automated Instrumentation and Monitoring System

The Automated Instrumentation and Monitoring System (AIMS) facilitates the tuning of parallel applications by capturing and visualizing execution data. To accomplish this, AIMS has three major software components: a *source code instrumentor*, a *run-time performance-monitoring library*, and a set of *visualization/analysis* tools for examining the collected data.

- The source-code instrumentor, **xinstrument**, inserts performance monitoring routines into the application's source code. The programmer can select from a menu those files and constructs that should be instrumented.
- The run-time performance monitoring library, or **monitor**, provides a set of monitoring routines that measure and record various aspects of program performance such as message passing overhead, synchronization overhead, and time spent in different subroutines.
- Two tools process and display the execution data. View Kernel (VK) animates the application's behavior on the multiprocessor; implementation bottlenecks and load imbalances can easily be observed. **tally** provides performance statistics for the entire program execution. These statistics provide insights into the general behavior of the program and may indicate where the animated views should be focused. The data from **tally** can also be used as input to statistical drawing packages such as GNUplot, WingZ or Microsoft Excel.

1.2. Background and History

AIMS was developed after evaluating several software prototypes from the research community and reviewing published ideas on performance visualization. We would like to acknowledge the community's help/support in letting us adopt, adapt and augment their research prototypes for the parallel-processing environment here at NASA Ames Research Center. The current version of AIMS uses the POEM source code instrumentation system developed under the Programming and Instrumentation Environment (PIE) project [Ref. 1] at Carnegie-Mellon University. AIMS' monitor adopts many of the event-record conventions established by PICL, a Portable Instrumented Communication Library [Ref. 2]. Some of the displays have been inspired by ParaGraph [Ref. 3] and Quartz [Ref. 4]. We also want to acknowledge the computing facilities provided to us by the Numerical Aerodynamic Simulation Division, NASA Ames Research Center. AIMS is developed under the sponsorship of NASA's High Performance Computing and Communications Program.

We also want to acknowledge the students who spent their summers working on various features for AIMS: Chris Hanson, Philip Tayco, Jacob Gotwals, and Brian Schmidt. Tarek Elaydi with Lawrence Berkeley Laboratories was responsible for building an AIMS prototype for TMC's CM5. Rob Gordon from Convex Computers Corp. was responsible for building an

AIMS prototype for PVM on UNIX platforms. Both versions will be available through COSMIC in future releases. Richard Papasin built `atopg`: a translator that converts our trace files to a format understandable by ParaGraph [Ref. 3].

1.3. System Capabilities and Requirements

AIMS 2.2 supports `Fortran77` and `C` under the `NX` (version 2/3.3.1, 3.3.2) operating system on the Intel iPSC/860. The instrumentation and visualization tools require a color workstation running `X Windows` (version `X11R5`) and `OSF/motif` (version 1.1.3); these tools have been tested on Sun SparcStations under Sun O/S 4.0.3 and on Silicon Graphics IRIS under `IRIX 3.3` (or 4.0) running the `twm` or `mwm` window manager.

1.4. Outline of Document

Chapter 2 introduces AIMS by running through an example and briefly describing its commonly used features. Chapters 3, 4, and 5, respectively, describe the instrumentor, the monitor, and the analysis tools in more detail. Chapter 6 explains how AIMS can be customized.

At the end of the manual are several technical appendices. Appendix A (Installation Guide) describes how to install AIMS. Appendix B (From Source Code to Trace Records) tabulates the source code constructs and corresponding trace records, along with the monitor routines that create the records and any necessary conditions for their creation. Appendix C (Trace Records) details the format and semantics of each trace record generated by the monitor. Appendix D (Customization Parameters) itemizes the parameters available for customization purposes. Appendix E (Converting AIMS trace files for ParaGraph) describes a simple tool that converts our tracefiles for ParaGraph.

2. Using AIMS — An Example

AIMS depicts the actual execution of a parallel program on a multiprocessor. The system's three main software components — an instrumentor (**xinstrument**), a run-time performance monitoring library (**monitor**), and a set of analysis tools (VK and **tally**) — measure and display a program's performance. **xinstrument** modifies the source code so that event timings and other information from the program's execution can be recorded. The **monitor** consists of a set of routines that are called by the instrumented code. These routines create a trace file used by VK and **tally**. VK provides a number of animated views that can be used to observe the program's behavior. **tally** records and tabulates cumulative statistics from the data in the trace file. Figure 2-1 below illustrates how **xinstrument**, **monitor**, and VK /**tally** interact.

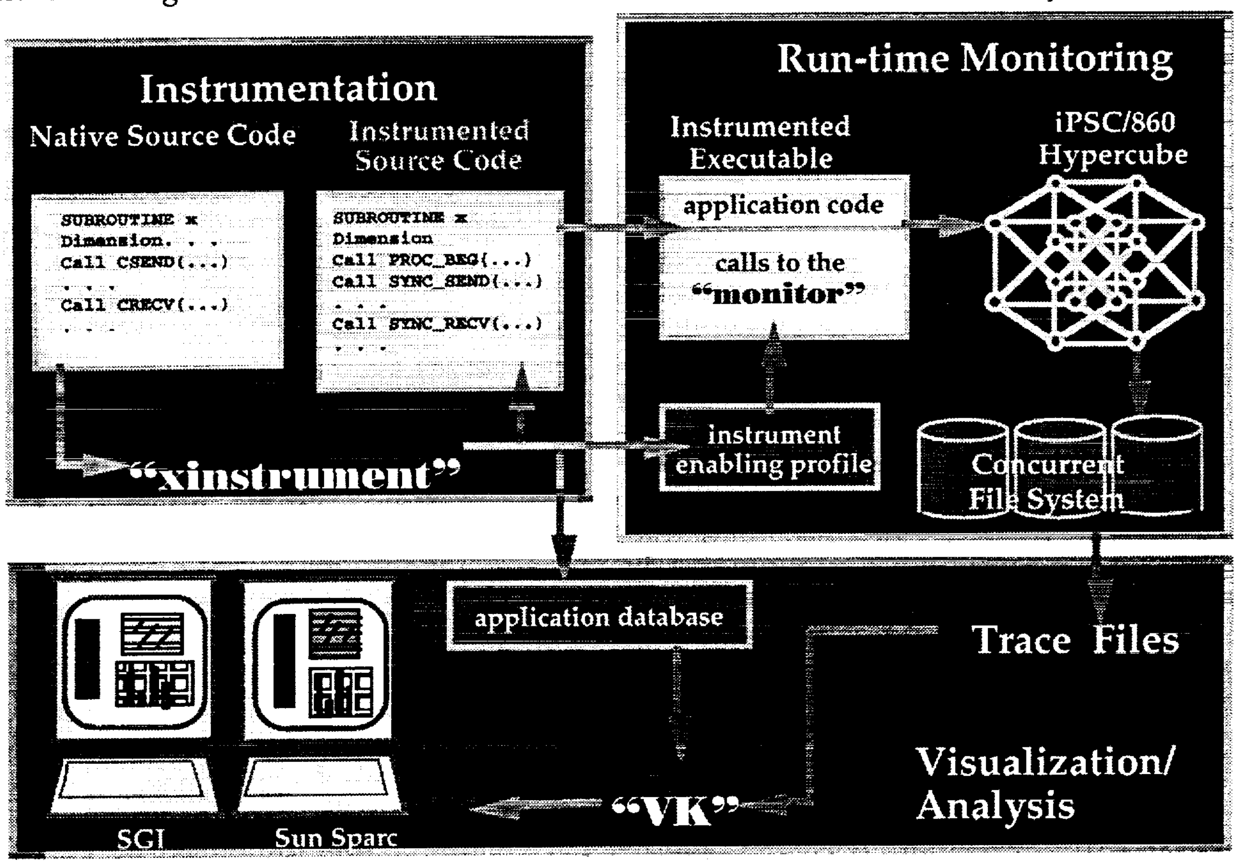

Figure 2-1. Using AIMS to Evaluate Parallel Program Execution

The user should note that in addition to inserting instrumentation at appropriate locations in the program code, **xinstrument** generates two important structures: an "application database" and an "instrument-enabling profile":

- The application database is used for storing information about the static structure of an application's source code. The analysis tools use this information in order to relate traced events to instrumented constructs in the source code. The instrumentation programs (see Section 3) build the application database and write it to a file that is subsequently incorporated at the beginning of the trace file produced by executing the instrumented application program.

- The instrument enabling profile is basically a table of flags, one for each construct in the application database. This profile is used twice:
  i) by the instrumentor to select the constructs to be instrumented; and
  ii) by the monitor to select the instrumented constructs to be traced.

Only those constructs whose flags are true in the profile will be instrumented/ traced; thus, the use of instrumentation can be switched on/off without recompiling the instrumented code! (see Chapter 3).

These three components of AIMS will be described in detail in Chapters 3, 4 and 5. We now give a quick overview of AIMS' ability to instrument, monitor and evaluate the performance of parallel programs.

2.1. The Transpose Program

A sample FORTRAN program is included with the AIMS distribution tape, in a top-level directory called `example/`. It contains six `Fortran77` files (five source code (`.f`) files and one include file (`transpose.incl`)), a `makefile`, a monitor option file (`.MONITOR`), two AIMS trace files (`TRACE.OUT` and `TRACE2.OUT`), and a `README` file. This parallel program "transposes" a matrix on the nodes of a hypercube. Initially, the matrix is distributed by rows among a certain number of processors; at the end of the run, the matrix is distributed by columns. The code provides two methods for achieving this. One uses "synchronous sends", and the other uses "asynchronous sends" with extra buffers. The user is prompted for the method to use, as well as the dimension of the hypercube and the number of matrix rows per processor.

In order to compile the host (`transpose_host`) and node (`transpose_node`) programs, the `makefile` may have to be modified (in particular, the lines specifying the location of the host and AIMS libraries). An execution log is shown in Figure 2-2; the program prompts for some information, and then prints a subset of the rows and columns of the matrix both before and after the transpose.

2.2. Source Code Instrumentation

xinstrument "instruments" application source code by inserting calls to *monitoring routines* into the code. These routines trace the performance of various constructs such as subroutine invocations, synchronization operations, and message routines. Typing "`xinstrument -overwrite`" from the source directory brings up a window as shown in Figure 2-3. There are three sections in the top-level window of **xinstrument**. They are, in the order of appearance, the *menu bar*, the *module table*, and the *instrument* button. On-line help for most windows can be displayed by pressing the MOTIF help key (usually `F1`). The module table lists all of the modules in the application database.

```
> transpose_host
 Enter dimension of cube:
3
 Enter number of matrix rows per processor:
40
 Enter method [1 or 2]:
1
 Transposing a(n)   320 x   320 matrix on a(n)   8-node cube.
 Allocating the cube...
 Loading the program onto the nodes...
 Nodes have been initialized.
        1   12801   25601   38401   51201   64001   76801   89601
       41   12841   25641   38441   51241   64041   76841   89641
       81   12881   25681   38481   51281   64081   76881   89681
      121   12921   25721   38521   51321   64121   76921   89721
      161   12961   25761   38561   51361   64161   76961   89761
      201   13001   25801   38601   51401   64201   77001   89801
      241   13041   25841   38641   51441   64241   77041   89841
      281   13081   25881   38681   51481   64281   77081   89881

        1      41      81     121     161     201     241     281
    12801   12841   12881   12921   12961   13001   13041   13081
    25601   25641   25681   25721   25761   25801   25841   25881
    38401   38441   38481   38521   38561   38601   38641   38681
    51201   51241   51281   51321   51361   51401   51441   51481
    64001   64041   64081   64121   64161   64201   64241   64281
    76801   76841   76881   76921   76961   77001   77041   77081
    89601   89641   89681   89721   89761   89801   89841   89881
 Transpose is complete.
>
```

Figure 2-2. Output from Matrix Transpose Example

Modules can be loaded into the application database by selecting the "`Load Modules`" item from the "File" menu. A Module Loader Window will appear, as shown in Figure 2-3. The user should select the "`F77 IPSC/860 Host`" platform for "`transpose_host.f`" and "`F77 IPSC/860 Node`" for the rest of the modules. The file names of the selected modules should now appear in the module table in xinstrument's main window. In order to trace the execution of different subroutines, select all the files in the *module table* before enabling the **Subroutine** option under the **Enable By Type** item from the **Profile** menu. All subroutines in selected modules will be instrumented when the **instrument** button is finally pressed. Pressing the **Exit** item will terminate xinstrument.

The instrumentor has now created an `inst/` directory containing the application database (a file named `APPL_DB`) and the instrumented source code (in this case, consisting of five FORTRAN files). The files `transpose.incl`,`makefile`, and `.MONITOR` should be copied to `inst/` from the source directory to complete the set-up.

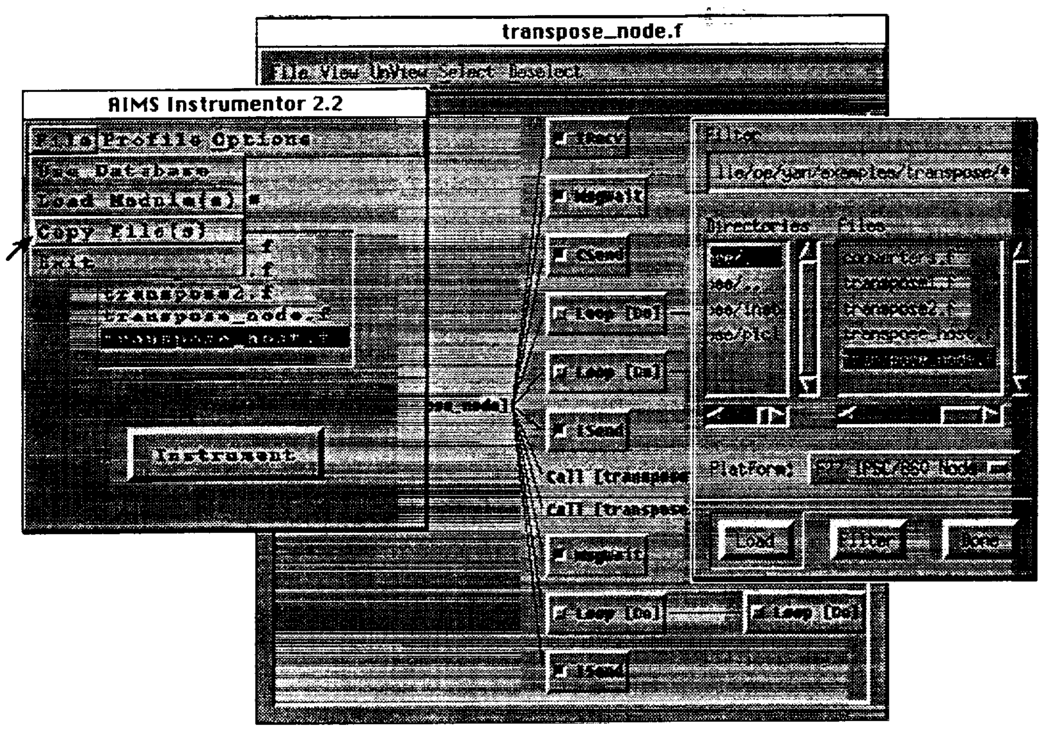

Figure 2-3. Graphical Interface of "xinstrument": AIMS's Source Code Instrumentor

2.3. Linking and Running

The second step in using AIMS is to link, compile, and run the instrumented code. The `makefile` has already been set up to link the AIMS libraries (`hostlib.a` and `nodelib.a`) with the transpose program. Compilation involves running "make" again, this time from the `inst/` directory.

The program runs just as before, except that information about certain monitor parameters is printed out, and a file called `TRACE.OUT` is created. This trace file is used as the input for the analysis programs.

Trace files (`TRACE.OUT` and `TRACE2.OUT`) included in the distribution was obtained by executing the transpose program on an 8-node cube, with 40 rows per processor using methods 1 and 2 respectively. If the user cannot use iPSC/860 for generating new trace files, the user may simply copy these into the `inst/` directory to try out VK.

2.4. Examining the Data

2.4.1. Creating a sorted trace file with **`tracesort`**

Trace files must first be sorted with the **`tracesort`** program by typing:

```
tracesort TRACE.OUT > TRACE.SORT
```

2.4.2. Trace file Animation with VK

To run the View Kernel, VK, "cd" to the (uninstrumented) source directory and type:

```
VK inst/TRACE.SORT
```

As shown in Figure 2-4, VK's main window has four VCR-like control-buttons that correspond to "*rewind*" ( ), "*play*" ( ), "*pause*" ( ), and "*single-step*" ( ) from left to right. Pressing these buttons has no effect unless one or more views are displayed. Clicking on **Views** allows the user to select one of many displays that VK provides. After opening the OverVIEW and Boxes (Circle) views, the user may hit the " " button to begin playback.

2.4.2.1. The OverVIEW

Figure 2-4. VK's Main Menu

The OverVIEW shows the procedures running on each node, and the messages being passed between them. As shown in Figure 2-5, the nodes are represented by rows; node numbers are listed on the left. OverVIEW uses colors or bitmaps to depict the various procedures. Thin lines on the OverVIEW indicate messages being sent between nodes. Rows of x's at the end of the "play-back" indicate that the nodes are writing the trace records to disk.

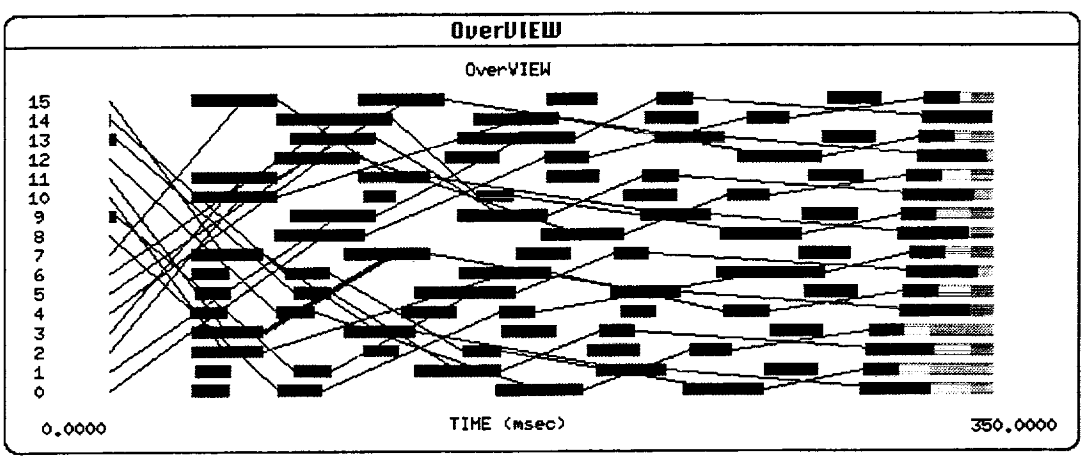

Figure 2-5. OverVIEW and the Application Legend

OverVIEW is initially set to scroll after 100 msec. of the trace file, so that it shows only 100 msec. worth of data at a given time. This value may be changed by typing "x" to the OverVIEW window to modify the length (in milliseconds) of the x-axis. Decreasing the scale will cause OverVIEW to show more detail, while increasing it will allow one to see more of the trace file at once. The value in the pop-up window can be edited with backspaces and keystrokes. When finished, the user should hit <Return>, or use the **OK** button at the bottom of the window. Changes will be immediately reflected in the OverVIEW. This process may be repeated by pressing the VCR Buttons corresponding to **Rewind** and **Play** from the main menu.

By clicking on the OverVIEW's window, one can obtain information about the constructs depicted there. For example, clicking on a procedure bar with the <u>middle</u> button will review the subroutine running at that time/node point. Holding the `<Shift>` key down while clicking will yield a window containing the code for the corresponding subroutine. Similarly, one can click on a message line with either the <u>left</u> or <u>right</u> buttons. Shift-clicking with the left button will show the code causing the send, and shift-clicking with the right button will show the code causing the receive. As shown in Figure 2-6, the exact line is pointed to by a "^" in the source-code window. Performing these mouse clicks with the `<cntrl->` key will produce "construct tree" views showing the relationship between the observed event and instrumented points in the source code.

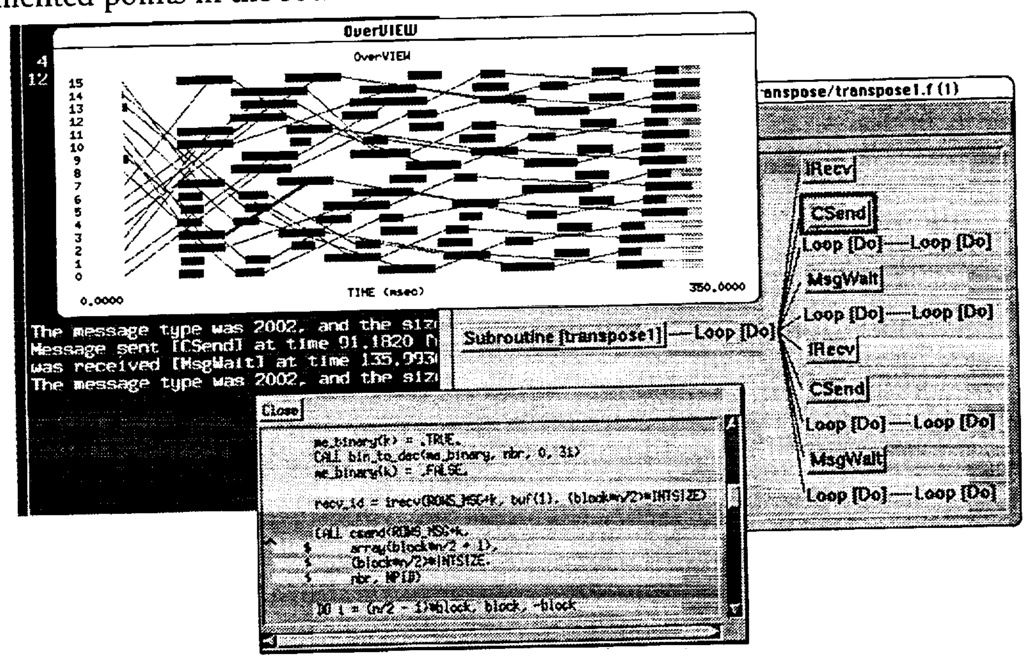

Figure 2-6. Relating Observed Events with the Source Code

Many features of the OverVIEW may be changed, including the order in which the nodes are listed and whether or not messages are displayed; the colors in which the procedures are drawn can be edited; and break-points in the display may be set. These features are explained further in Chapters 5 and 6. One can also press the MOTIF help key (usually `F1`) in OverVIEW's window for help.

2.4.2.2. The Boxes Views

The Boxes views depict the state of each processor and the messages passing between the processors. The Grid version is meaningful only if the topology underlying the algorithm is a grid. In that case, "`define_node`" and "`define_grid`" calls must be inserted into the code to activate the Grid view. The Circle version works without these calls.

As shown in Figure 2-7, each processor is represented in the Boxes Views by a rectangle containing five small boxes. The three central boxes contain the node number, node state, and current procedure, from top to bottom. The *Node States Legend* (click on **Legends** in the main menu, then on **Node States**) shows the possible node states. The column to the left of the node number and state indicates the number of messages pending for the node. (A *pending message* is a message that has been sent but not yet processed by the receiver.) A full column means 10 or more messages are pending. The column to the right indicates the *node utilization*, the proportion of execution time the node has spent doing useful work. A low value indicates that the node has been blocked for a large amount of time. Lines between boxes indicate the presence of one or more pending messages.

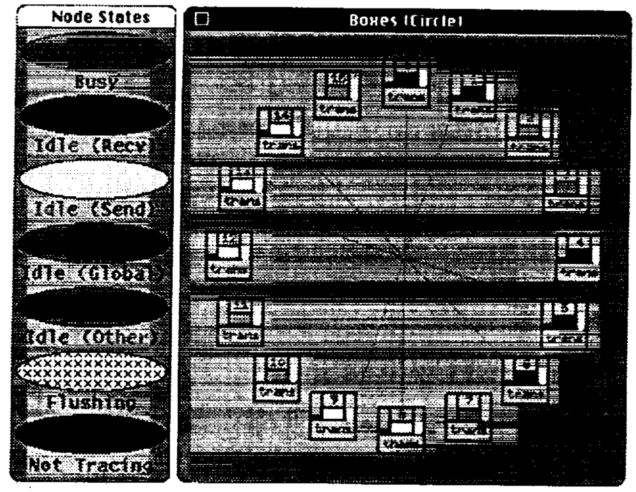

Figure 2-7. Boxes (Circle) View and Node State Legend

2.4.2.3. Pausing VK

Since some of VK's views (such as the Boxes views) are not scrolling views, it can be useful to pause in the middle of the trace file so the display will reflect the program's state at that point in time. With VK, the user can choose either to pause at a certain *time* in the trace file, or to pause when one or more *subroutines* are reached. This is done via the "Time Control" option under the **Controls** menu.

Setting the "pause" and "resume" times will cause VK to pause and/or resume reading the trace file when it first reaches a record occurring on or after the specified time. The values in the Time Control Window may be changed by placing the cursor over the value to be changed. Unless the `<Return>` key is pressed when one is done editing; changes will not take effect. Figure 2-8 was obtained when VK pauses at time 36.76 while viewing `TRACE.OUT`.

VK may be paused at a certain instrumented construct via the **Breakpoints Enabled** option under the **Control** menu. In order to identify the break point, specific files containing the source code have to be selected from the **Construct Legend** under the **Control** menu. Figure 2-9 illustrates this situation with the transform example. Now, if one resets and views the trace file, VK will stop whenever any node encounters one of the selected constructs. Hitting the "▶" button in the main window will resume playback.

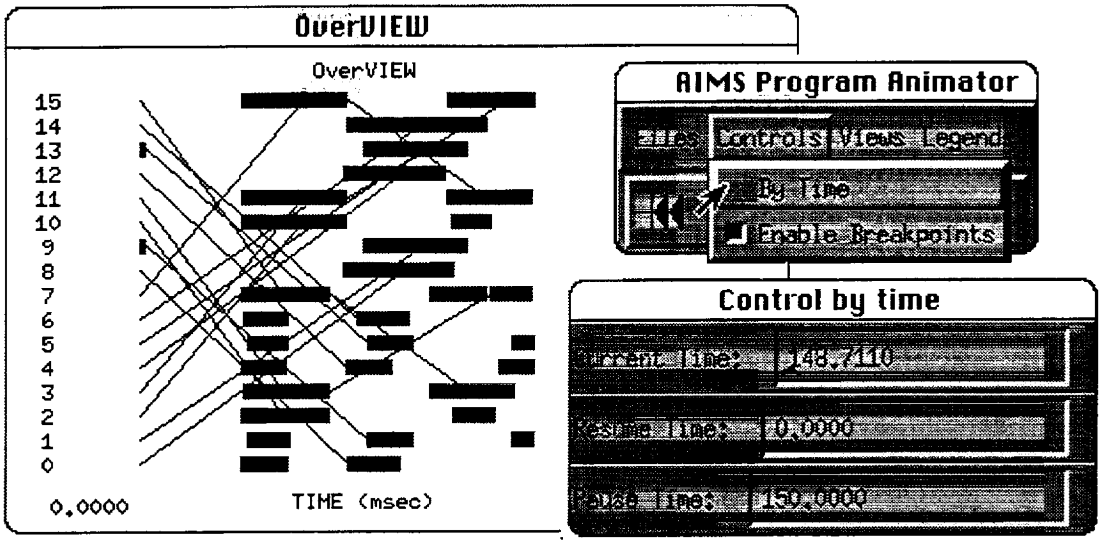

Figure 2-8. Time Control Window

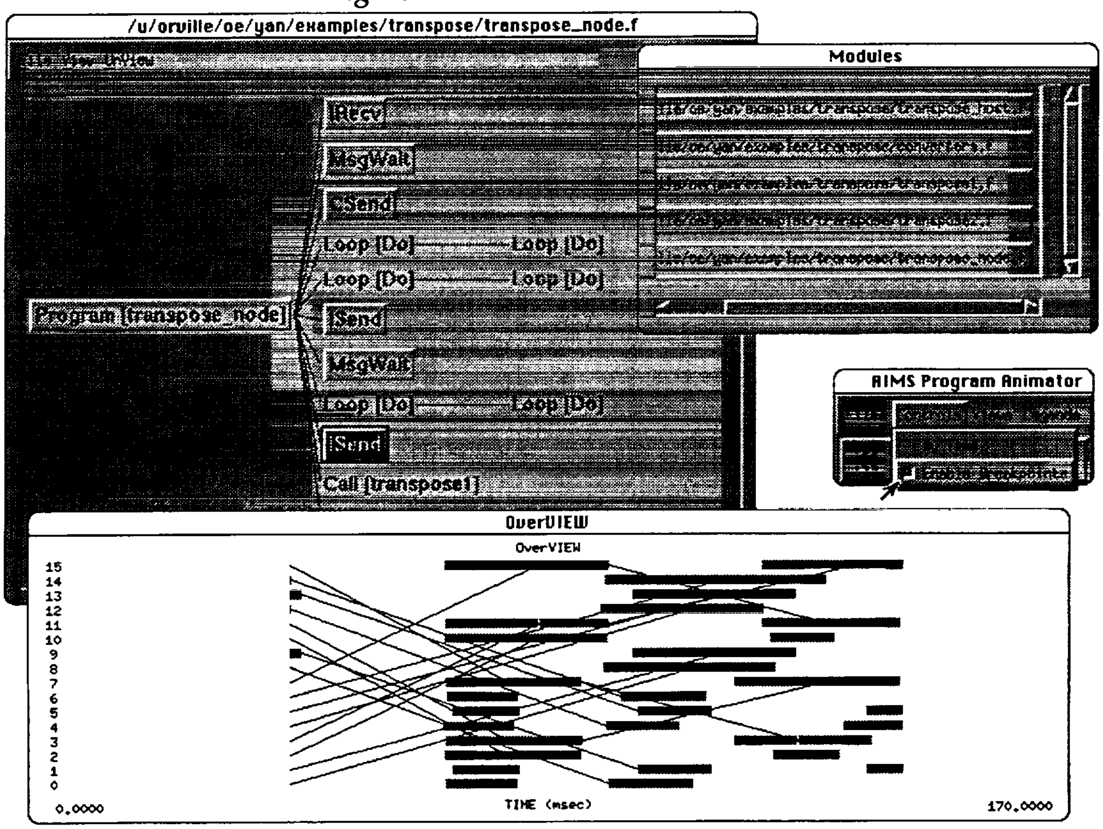

Figure 2-9. Break-point Control Window

VK has a number of other useful views, as described in Chapter 5. But VK does not provide cumulative statistics; these are provided by the tally program.

2.4.3. Performance Summary with `tally`

For each node, **tally** collects the following statistics: lifetime, busy time, volume of messages received, amount of time spent blocking on a receive, volume of messages sent, amount of time spent blocking on a send, number of times in a global operation, and the amount of time spent in global operations. These raw data can be seen by typing "`tally inst/TRACE.SORT | more`" from the source directory. In addition, **tally** provides some statistics (see Section 5.3) which help in focusing on problem functions and processors.

tally also breaks these statistics down by subroutine, so one can know, for example, as shown in Figure 2-10, how much time was spent blocking on receives by node 4 in subroutine `transpose1`. These statistics are printed out in tables on both a per-node and per-subroutine basis, as described in Section 5.3.

2.5. Summary

The reader should now have an idea of how AIMS works. The reader should familiarize him/herself with VK's other views, read the help windows for information about them, and try to use AIMS to compare methods 1 and 2 for transposing a matrix.

The remainder of the manual discusses the system in more detail: Chapter 3 discusses the instrumentor; Chapter 4, the monitor; and Chapter 5 the analysis tools. Chapter 6 explains the different ways to customize AIMS.

Tables for trace file 'T':

ROUTINE SUMMARY

Routine	Busy Time	Global Blocking	Send Blocking	Recv Blocking	Life Time	% Commn	Comm. Index
1 transpose1	229.399	0.000	246.186	64.887	540.472	57.555	0.387
2 transpose_node	100.659	0.000	0.186	160.826	261.671	61.532	0.200
3 bin_to_dec	0.508	0.000	0.000	0.000	0.508	0.000	0.000
4 dec_to_bin	0.128	0.000	0.000	0.000	0.128	0.000	0.000
5 <rest..>	0.117	0.000	0.000	0.000	0.117	0.000	0.000

NODE SUMMARY

Node	Busy Time	Global Blocking	Send Blocking	Recv Blocking	Life Time	% Commn
0	37.443	0.000	34.241	28.154	99.838	62.496
1	39.927	0.000	35.575	28.342	103.844	61.550
2	39.560	0.000	24.994	33.395	97.949	59.611
3	44.796	0.000	29.530	27.495	101.821	56.005
4	39.289	0.000	30.399	28.216	97.904	59.869
...						

STATISTICS FOR ROUTINE transpose1

Node	Busy Time	Global Blocking	Send Blocking	Recv Blocking	Life Time	% Commn
0	24.754	0.000	34.220	8.191	67.165	63.144
1	27.263	0.000	35.555	8.232	71.050	61.628
2	26.861	0.000	24.967	13.341	65.169	58.782
3	32.139	0.000	29.509	7.381	69.029	53.441
4	26.605	0.000	30.374	8.156	65.135	59.154
...						

STATISTICS FOR ROUTINE transpose_node

Node	Busy Time	Global Blocking	Send Blocking	Recv Blocking	Life Time	% Commn
0	12.599	0.000	0.021	19.963	32.583	61.332
1	12.573	0.000	0.020	20.110	32.703	61.553
2	12.605	0.000	0.027	20.054	32.686	61.436
3	12.562	0.000	0.021	20.114	32.697	61.580
4	12.589	0.000	0.025	20.060	32.674	61.470
...						

NCPU STATISTIC

Routine	1	2	3	4	5	6	7	8
transpose1	5.342	13.957	7.038	9.436	5.806	8.677	5.325	2.363
transpose_node	0.289	0.017	0.091	0.110	0.063	0.062	0.042	12.329
bin_to_dec	0.000	0.012	0.029	0.032	0.014	0.015	0.008	0.005
<rest..>	0.004	0.003	0.001	0.001	0.001	0.001	0.004	0.006
dec_to_bin	0.000	0.000	0.000	0.000	0.005	0.001	0.004	0.007
<flush>	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

ROUTINE CONCURRENCY STATISTIC

Routine	1	2	3	4	5	6	7	8
transpose1	5.667	14.060	7.109	9.631	5.914	8.584	5.257	2.236
transpose_node	0.599	0.171	0.363	0.006	0.057	0.058	0.092	12.166
bin_to_dec	0.386	0.061	0.000	0.000	0.000	0.000	0.000	0.000
dec_to_bin	0.050	0.039	0.000	0.000	0.000	0.000	0.000	0.000
<rest..>	0.038	0.002	0.000	0.000	0.000	0.000	0.005	0.005
<flush>	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Figure 2-10. Excerpt from Tally's Output for Transpose

3. Source Code Instrumentation

AIMS' instrumentors insert calls to monitoring routines into code. Such calls replace or surround instrumentable constructs, such as message sends and receives, synchronization operations and subroutine invocations.

AIMS provides two instrumentors, **xinstrument** and **instrument**, for instrumenting source code. **xinstrument** is window-based, while the simpler **instrument** is not. The latter is, therefore, useful in makefiles and shell scripts.

The **xinstrument** and **instrument** commands are described in details in the next two sections. The third section describes *instrumentor directives* that can be inserted into source code by hand in order to direct the automatic instrumentation. A concise summary of the information presented here is available as on-line help for **xinstrument**.

3.1. xinstrument

The **xinstrument** program instruments source code with minimal user-intervention. An X-based interface facilitates the instrumentation of source files as well as of program constructs within those files. These instrumentor specifications, called *profiles*, can be saved for later use.

3.1.1. Calling xinstrument

xinstrument is invoked as follows, from the directory containing the source code:

```
xinstrument  [-adb <application database>]
             [-oadb  <output application database>]
             [-overwrite]
             [-help]
             [-output <directory>]
             [-origin <directory>]
             [-platform <option>]
             [-run_pp]           (run the preprocessor)
             [-no_pp]            (do not run the preprocessor)
             [-pp_command <command-template>]
             [-pp_options <switches>]
             <file(s)>
```

The application database is used by **xinstrument** to determine which files to consider for instrumentation. The output application database is used to store information about those files and is later used by the analysis tools to relate the trace file to the source code. If no database is specified on the command-line, **xinstrument** will use the file APPL_DB in the output directory. In some cases, the output directory may already exist; **xinstrument** takes a precautionary measure and does not overwrite an existing file that already exists unless the -overwrite flag is specified.

xinstrument places the instrumented code in the output directory. Note that uninstrumented files, such as makefiles and include files, are not written into this directory by the instrumentor, and must be copied over manually after the directory has been created. If no output directory is specified, the directory `./inst` is used.

xinstrument does not require that all source files be found in the same directory. Thus, if multiple paths are utilized, the **-origin** option can be used to define the root of a multi-directory source tree. If no root is explicitly specified, the current working directory is used by default.

If any files are specified on the command-line, **xinstrument** loads them into the application database using the platform defined with the `-platform` flag. These files can be instrumented along with any others that may already have been stored in the database. The platform options are `f77-nx-host`, `f77-nx-node`, `c-nx-host` and `c-nx-node`. If the file to be loaded is a FORTRAN file and should be run as a host program, the `f77-nx-host` option should be used. Similarly, the `f77-nx-node` option should be utilized for FORTRAN node programs; `c-nx-host` for C host programs; `c-nx-node` for C node programs. If a `-platform` flag is not used, the platform is assumed to be `c-nx-host`. Files should be reloaded into the database whenever they are modified. While it is possible to instrument a newly modified file without reloading it into the database, this may cause AIMS to fail.

If the code to be loaded does not need to be run through a preprocessor, the `-no_pp` flag should be used. Otherwise, the user should first define the preprocessing command and options using the `-pp_command` and `-pp_options` flags and then use the `-run_pp` flag. AIMS assumes that the preprocessing command sends the "preprocessed" file to stdout. Thus, preprocessing commands that do not utilize this can not be used.

If the `-help` flag is present, **xinstrument** prints a usage message and exits.

xinstrument will also accept standard X options such as `-fn` (font), `-bg` (background color), and `-fg` (foreground color).

3.1.2. Using xinstrument

There are three sections in the top-level window of **xinstrument**. They are, in the order of appearance, the *menu bar*, the *module table*, and the *instrument* button.

The Files menu contains four options: **Use Database**, **Load Module(s)**, **Copy File(s)** and **Exit**. The **Use Database** option allows a previously defined application database to be loaded into the modules list. The loading of this file containing the database is done through a file selection window. A file is selected by clicking on the appropriate file name with the left mouse button. Once the menu item is highlighted, it can be loaded by pressing OK or <return>.

Using the **Load Modules** file selection window, individual modules can be loaded into a list of modules to be instrumented. Single clicking the left (CTRL-left) mouse button or dragging it will select (deselect) files. The desired platform is selected under the **Platform** menu; **F77 IPSC/860 Host** is used for FORTRAN host programs, **F77 IPSC/860 Node** is used for FORTRAN node programs, **C IPSC/860 Host** is used for C host programs, and **C IPSC/860 Node** is used for C node programs. Once all files have been selected, loading will take place if the **Load** or `<return>` button is used. The **Done** button is selected to dismiss the window.

The middle panel of the main window shows all of the files in the application database. Files can be selected by clicking the left mouse button on top of the file name, and multiple files can be selected using CTRL-left mouse button; the selected files are highlighted, and will be instrumented when the **Instrument** button is clicked. Within each file are a number of instrumentable constructs. One can select the constructs to be instrumented by double clicking the left mouse button on the file of interest. This brings up a list of all instrumentable constructs for that file, as shown in the window titled `transpose_node.f` of Figure 3-1. Clicking on constructs will select (or deselect) them; selected constructs are highlighted, and will be instrumented if the file is instrumented†. By default, only system-level constructs, such as message routines and global operations, will be selected for instrumentation. Subroutines will not be selected. Note that the selected constructs will be instrumented only in the file currently selected for instrumentation.

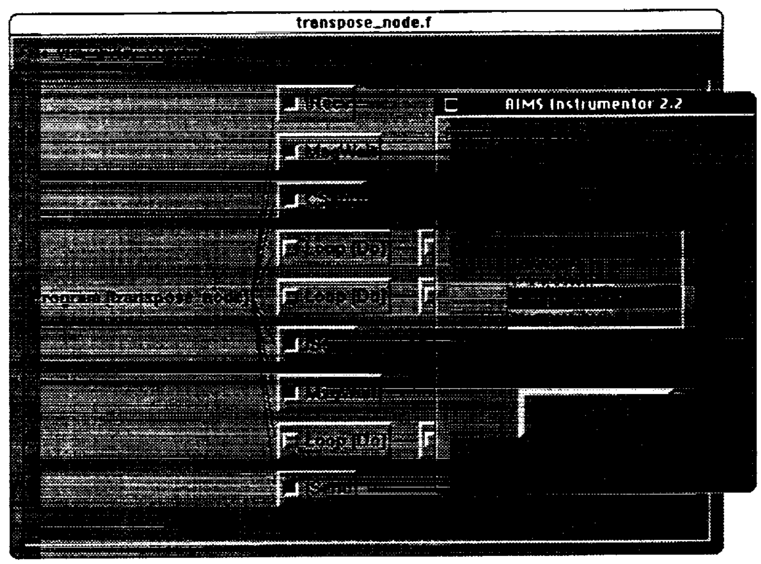

Figure 3-1. xinstrument

The top three options in the Profile Operations panel allow one to select all constructs, deselect all constructs, or select only the default constructs of the application (as specified by AIMS). The next two options are used to enable or disable constructs by type. These operations will apply to all of the files highlighted in the Files panel. With the **Save** and **Load** buttons, one can store and reload instrumentation profiles. When either of these buttons is pressed, a dialogue box will appear requesting the user to enter the name of the file containing the profile. Hitting **Save** or **Load** or `<return>` will save or load the file, while **Cancel** will discontinue the operation.

† Certain constructs will *always* be instrumented, such as program starts and terminations. Others will be instrumented only if related constructs are instrumented, such as subroutine returns.

The third menu contains five user-settable options: the *output directory*, the *origin directory*, the *run preprocessor selection*, the *set preprocessor command*, and the *set preprocessor options*. The output directory specifies where the instrumented files will be saved. Note that changes to the output directory will not take effect until **OK** or `<return>` has been pressed in that dialogue box. If the files require preprocessing before loading the modules, the user must give the preprocessing command by selecting the **Set Preprocessor Command** option and then selecting the **Run Preprocessor** option of the menu.

Pressing **Instrument** causes all selected files to be instrumented. The resulting files will be saved in the output directory with the same name as the original source file. The **Exit** button under the File Menu terminates the application.

3.2. Batch mode instrumentors

The instrument program is a simplified version of the **xinstrument** program. It is not X-based, and can therefore be useful in makefiles and shell scripts. The appropriate binary changes depending on the desired platform. The name of the binary files follow the form:

inst-<platform>;

where <platform> is either **c-nx-host**, **c-nx-node**, **f77-nx-host** or **f77-nx-node**.

3.2.1. Command-line syntax

The command-line syntax of using the batch mode instrumentor is:

```
inst-<platform> [-adb <application database>]
                [-oadb <output application database>]
                [-overwrite]
                [-origin <directory>]
                [-output <directory>]
                [-help]
                [-run_pp]          (run the preprocessor)
                [-no_pp]           (do not run the preprocessor)
                [-pp_command <command-template>]
                [-pp_options <switches>]
                [-verbose]
                [-enable <option>]
                <file(s)>
```

The profile option is `all`, `default`, or the name of a file containing a stored profile.

The `-adb`, `-oadb`, `-output`, `-origin`, `-overwrite`, `-run_pp`, `-no_pp`, `-pp_command`, `-pp_options` and `-help` switches function as with **xinstrument** If the `-verbose` flag is present, status messages are printed.

The `-enable` option is used to specify which constructs are instrumented. By default, AIMS instruments only system-level calls, such as message routines and global operations, and no subroutines. If the `all` option is specified, all constructs are instrumented. (Note that selecting `all` may generate a VERY LARGE tracefile!) If the option is neither `default` nor `all`, the instrumentor expects the name of a profile file. (Profile files are generated by **xinstrument**, as described in Section 3.1.2.)

The `-enable` switch may appear more than once on the command-line; each file will be instrumented using the switch that most closely precedes it. At least one source file should be specified on the command-line.

3.2.2. Differences with xinstrument

The batch mode program instruments only those files listed on the command-line. Therefore, at least one file should be specified on the command-line. **xinstrument**, however, looks for files in the application database, as well as on the command-line, so it is possible to use **xinstrument** without specifying any files at all.

The instrumentor instruments all of the files that are specified on the command-line. **xinstrument** loads files into the database, but one can then choose not to instrument some files.

The batch mode instrumentor is not X-based, so it has platform specific calls and `-enable` switches to perform the functions of some of **xinstrument**'s buttons. Also, because it is not X-based, it can easily be called from makefiles and shell scripts.

3.3. Instrumentor Directives

The automatic instrumentation provided by **xinstrument** may not be sufficient for your purposes. You may want to instrument constructs that **xinstrument** does not recognize, or you may find that you only need to instrument only a few rather than all of the iterations of a loop. To handle these and similar situations, AIMS provides eight *instrumentor directives*. The instrumentor replaces these directives with calls to monitor routines, just as it replaces calls such as `csend`.

3.3.1. Using Instrumentor Directives

Instrumentor directives may be simply inserted into a program as subroutines. For example, the line

```
CALL insert_marker('Starting loop')
```

marks the beginning of a loop. When the program is executed, the corresponding monitor routine will generate a MARKER trace record, which will be displayed by VK's OverVIEW.

The user can compile and run uninstrumented source code containing instrumentor directives if the code is linked with the monitor libraries, as described in Section 4.2. The routines have no effect in uninstrumented code.

3.3.2. **begin_trace** and **end_trace**

The begin_trace and end_trace directives allow you to turn tracing on and off. While the automatic instrumentor allows only that certain selected constructs be always traced, it doesn't permit one to specify general conditions for tracing. On the other hand, inserting begin_trace and end_trace directives (e.g., inside IF statements) allows one to trace only when something unusual happens, or when some value gets particularly large. These directives can be inserted anywhere you can place a CALL statement. If two begin_trace's (end_trace's) are called without an intervening end_trace (begin_trace), the second call has no effect.

3.3.3. **begin_block** and **end_block**

The begin_block and end_block directives allow you to treat a segment of a subroutine as a subroutine in its own right. The analysis tools will collect data for this region just as for any subroutine. You can use these directives either to provide more detailed monitoring by instrumenting small blocks within an instrumented subroutine or, less detailed, by creating blocks around calls to subroutines and deselecting those subroutines so that they are not instrumented.

These directives take a single string (CHARACTER*n) argument, which is the name you want to associate with the block. For example, you might insert calls to begin_block ('Initialize') and end_block ('Initialize') around an initialization section. The string arguments in a pair of calls should match.

Blocks may be nested, but they should not overlap. In general, blocks should be structured in such a way that the statements inside could be replaced by a subroutine. This means, for example, that constructs such as the following are *not* allowed. (Note that such placements *are* allowed for begin_trace and end_trace.)

```
CALL begin_block('Bad Block')
IF (i .EQ. 1) THEN
   ...
   CALL end_block('Bad Block')
   ...
END IF
```

3.3.4. **insert_marker**

The insert_marker directive can be used to get timing information at any point in the code where you can call a subroutine. This directive takes a single string (CHARACTER*n) argument, which is the name of the marker. The marker will be displayed by VK's OverVIEW as a vertical line in a procedure bar (see Section 5.2.1). OverVIEW will also print out the time each marker occurred.

3.3.5. **flush_trace**

The `flush_trace` directive allows you to indicate when you want the processor to write its trace records to disk. As described in Section 4, each node stores the records generated by the monitoring routines to a buffer in the node's memory; this buffer is periodically *flushed* (written to disk). While the monitor will automatically do a flush when the record buffer fills up, that strategy may cause flushing to occur at a time that will significantly disturb the program's execution. Inserting `flush_trace` directives will help to prevent undue perturbation, especially when the record buffer is relatively small. A `flush_trace` can be inserted anywhere a `CALL` statement can appear.

3.3.6. **define_grid** and **define_grid_node**

If your algorithm has an underlying grid topology, the `define_grid` and `define_grid_node` directives can be used to enable VK's Boxes (Grid) view. That view displays processor nodes in a rectangular grid, with each intersection occupied by a single node. The `define_grid` directive specifies the dimensions of the grid, and `define_grid_node` specifies the location of a node on that grid.

The `define_grid` directive takes two positive integer parameters, indicating the number of rows and columns, respectively, in the grid: "`CALL define_grid(rows, cols)`". The `define_grid_node` directive, which must be called from a node, also takes two integer parameters. They specify the coordinates of the calling node on the grid: "`CALL define_grid_node(i, j)`", where 1 <= *i* <= rows and 1 <= *j* <= cols.

3.4. Limitations of the Instrumentors

3.4.1. Using Labels

The instrumentors do not account for labels when inserting instrumentation. This can cause problems. For example, when the instrumentor sees a FORTRAN END statement, it inserts a few monitor routines before that statement (as shown in Example 3-1).

Source Code	**Instrumented Code**
`    send_id = isend(...)`	`    send_id = async_send(...)`
`  END IF`	`  END IF`
`END`	`CALL stop_trace`
	`CALL mon_term`
	`CALL proc_end(3,0)`
	`END`

Example 3-1: Instrumentation of an END Statement

If a label precedes the END statement, these calls are *not* inserted between the label and the END statement, but rather after the line preceding the END statement. This can result in the instrumented program hanging, since a GOTO statement elsewhere in the code will go right to the END statement without executing these routines.

Source Code	Instrumented Code
`      send_id = isend(...)`	`      send_id = ...`
`    END IF`	`    END IF`
`10 END`	`   CALL stop_trace`
	`   CALL mon_term`
	`   CALL proc_end(3,0)`
	`10 END`

*Example 3-2: Faulty Instrumentation of a Labeled **END** Statement*

As a result, we recommend that labels be used with CONTINUE statements, with code continuing on subsequent lines.

Source Code	Instrumented Code
`      send_id = isend(...)`	`      send_id = ...`
`    END IF`	`    END IF`
`10 CONTINUE`	`10 CONTINUE`
`   END`	`   CALL stop_trace`
	`   CALL mon_term`
	`   CALL proc_end(3,0)`
	`   END`

*Example 3-3: Using a **CONTINUE** Statement with a Label*

3.4.2. Warning Messages During Compilation

The instrumentor inserts instrumentation conservatively, which may result in the placement of instrumentation calls in positions where they will never be reached. For example, the three calls inserted in the previous example are inserted before both STOP and END statements. If a STOP precedes an END statement in the source code, the three calls following the STOP will never be reached. The compiler may warn you in such cases about the presence of unreachable statements. They will not cause an error, however.

3.4.3. Warning Messages During Execution

At the end of an instrumented host program's execution, a message indicating that the host is doing an illegal iprobe may appear:

```
(host) iprobe: No pid defined
```

This is again due to conservative insertion. Since it happens only after the monitor has finished tracing, the erroneous probe does no harm. Sometimes at the beginning of an instrumented host program's execution, the same error may occur but will stop execution. This happens when a construct (usually a loop, but sometimes a subroutine) is instrumented before a call to `load`. When this occurs, re-instrumentation of the code without those constructs selected will fix the problem.

3.4.4. What is Not Instrumented

The constructs that are instrumented are listed in Appendix B. Note that `cprobe`, `csendrecv`, `gcol`, `gcolx`,`gsendx`, `hsend`, `hsendrecv`, `isendrecv`, `killproc`,`msgcancel`, `waitall`, and`waitone`, among other constructs, are *not* instrumented automatically. To record information about those routines, you can use the `insert_marker` and `block_begin/end` instrumentor directives.

3.5. Preprocessing Programs

C programs (and to a lesser extent, FORTRAN programs) rely heavily on the preprocessor to resolve macro definitions before the actual parser is applied to the source code. Since the instrumentors only have native parsers in them, it may be necessary to pass the source code through a preprocessor stage in order for the code to be recognizable by the parsers. To this end, AIMS provides mechanisms by which the user can specify how (as well as if) a file should be preprocessed.

The preprocessor invocation is specified as a straight command (minus the file names). Since the invocation may differ for host programs and node programs, there can be separate templates for each of these cases. For example, the invocation for a C preprocessor on a node module might be: `icc -E`. The command may be specified in an environment variable, on the command line, under the **Options** menu of **xinstrument** or as the default given at compile time. Two environment variables are recognized by AIMS instrumentors (the instrument programs and **xinstrument**): `AIMS_PP_COMMAND` & `AIMS_PP_OPTIONS`. In addition, AIMS' instrumentors can accept these definitions on the command line using the following switches:

```
-pp_command <preprocessor command>
-pp_options <switches>
```

To specify the preprocessor command as the default one given at compile time, the file `Makefile.template` in the `instrumentor's` directory should be modified. Within this file, under the heading for the desired platform (`f77-nx-node`, `f77-nx-host`, `c-nx-node` or `c-nx-host`) are the platform specific variables: `PP_COMMAND` and `PP_OPTIONS`. In all cases, definitions given on the command line override those specified in environment variables which, in turn, override their default values.

4. Run-Time Performance Monitoring Library

The run-time performance monitoring library contains a set of routines which function as "event recorders" by measuring the behavior of the instrumented program on a parallel machine. These event recorders are placed by the instrumentor into the instrumented code; they are responsible for generating a trace file to be used by the analysis tools. Figure 4-1 illustrates the basic function of event recorders during the monitoring phase using the example from Figure 3-1. Note that the event recorders write records into a buffer that is only intermittently written to disk. Since writing the buffer to disk is a time-consuming process that can affect the execution pattern of the application, AIMS provides a number of ways to specify the size of the buffer and how often the buffer is emptied. The following sections describe the monitor's parameters, and explain the process of linking the monitor's routines with instrumented code.

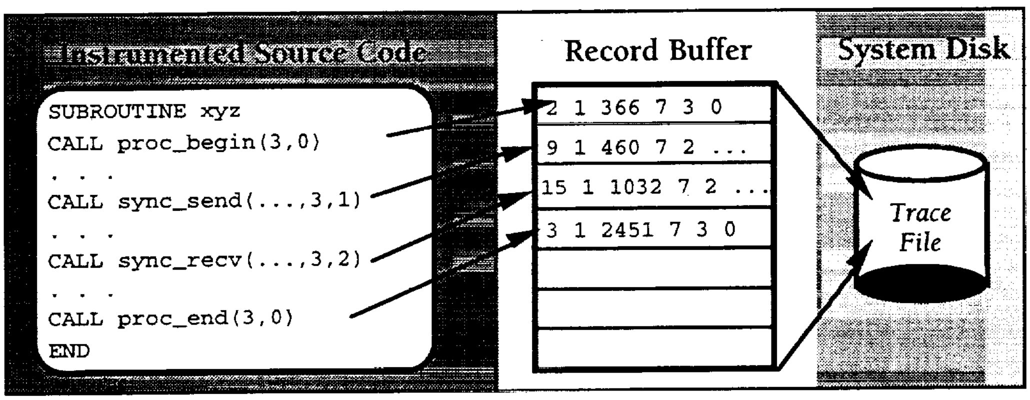

Figure 4-1. Inserted Event Recorders Generating Trace Records

4.1. Monitor Parameters

The operation of the monitor is controlled by several user-settable parameters, such as the name of the trace file and the size of the monitor buffer. These parameters are set in the initialization file `.MONITOR` which must reside in the directory from which the instrumented application is launched. Although all the parameters described below have default settings, they may not be appropriate for all applications. Experimentation may be required since it is sometimes difficult to exactly predict what level of monitoring is needed. Any subset of these options may appear in any order in the `.MONITOR` file. A section explaining how to set their values follows.

4.1.1. **TRACE_FILE**

All generated events are ultimately written to a disk file. The `TRACE_FILE` parameter specifies the name of that file. Because trace files can be several megabytes long, it is important to place the file where there is enough storage. If the iPSC/860 Concurrent File System (CFS) is the one of choice, this trace file will only contain data from the processor nodes; the host data will be written to a different file (see the `H_TRACE_FILE` parameter). The default file name depends on other settings in the configuration. If the application is running hostless or if the file mode (see below) specifies to use the CFS, then the default is /cfs/$USER/TRACE.OUT; otherwise, it is ./TRACE.OUT.

4.1.2. **H_TRACE_FILE**

The file specified by the H_TRACE_FILE parameter contains the events generated by the instrumented program run as the host. This file is used only when there is a host program (see below), and the nodes have been directed to use the CFS. The default file name for H_TRACE_FILE is HTRACE.OUT.

4.1.3. **FILE_MODE**

The FILE_MODE parameter specifies the file system being used to store the trace data output by the monitor. A value of 1 indicates the CFS file system, which is the fastest way to write large trace data files to disk. A value of 0 indicates that the nodes should write to a file system attached to the front-end host.

The nodes write to the front end by sending messages to the host; the host then writes the records to disk†. Specifically, when a node flushes its record buffer, it repeatedly calls csend to transmit small blocks of data to the host until its buffer is empty. The host repeatedly probes for these messages and writes the data to disk as they are received.

Writing to the CFS is typically much faster than the process for the frond end described above; thus, the use of the CFS is recommended. Note that if the CFS is used (i.e., FILE_MODE = 1), the file specified in the TRACE_FILE variable must have the appropriate CFS path in it.

4.1.4. **HOST_PROGRAM**

The HOST_PROGRAM parameter informs the monitor whether the application has an instrumented host program. The default value for this parameter is 0, which indicates no host program is being used. If this parameter is not set to 1 when a host program exists, the monitor initialization process may fail. Note that hostless programs *must* locate the trace file on the CFS, as described in the previous section.

4.1.5. **TRACE_LEVEL**

The TRACE_LEVEL parameter determines when the monitor will generate trace records. The lower is the value of this parameter, the fewer are the records produced. There are four possible levels, the default being level 3:

level 0 — generate TRACE_BEGIN, TRACE_END,BLOCK_BEGIN, BLOCK_END, and MARKER records

level 1 — generate level 0 records + PROC_BEGIN and PROC_END records

† Note that hostless programs therefore need to have their trace files on the CFS.

level 2 — generate level 1 records + GLOBAL_BEGIN and GLOBAL_END records

level 3 — generate level 2 records + message-passing records

Since the default level is 3 , the monitor generates all records unless specified otherwise. Some records will be generated even if a negative level is specified: MON_BEGIN, MON_END, FLUSH_BEGIN, FLUSH_END, DEFINE_GRID, and DEFINE_GRID_NODE.

Appendix B lists the monitor's event recorders (with the source code constructs they replace or surround) and the trace records they produce.

4.1.6. **BLOCK_ON_ALL_SYNC**

If the value of the BLOCK_ON_ALL_SYNC parameter is non-zero, the monitor will generate a pair of records for each call to csend and crecv, one at the beginning and one at the end. This allows the analysis tools to determine the length of time the node was blocked on those routines. If the value is 0, however, the monitor will generate only one record for a csend, indicating the start of the send. Thus the analysis tools will not count the time spent inside the csend as blocking time, but as busy time. In addition, the monitor will execute an iprobe before each crecv, and if the probe indicates that the message is present, it will generate only one record for the crecv at the completion of the call. Thus the time spent in such crecvs will also be considered busy, rather than blocked.

A zero value will result in a smaller trace file than a non-zero value, but will cause the monitor to do some extra probes (prior to crecvs), and the analysis tools will not indicate blocking on csends and quick crecvs. Thus a non-zero value is likely to be a good choice when the trace file is not too big and when the time spent blocking on csends may be significant.

The default for BLOCK_ON_ALL_SYNC is 1.

4.1.7. **BUFFER_SIZE**

The BUFFER_SIZE parameter sets the number of bytes to be allocated for the temporary storage of trace data on each node. It is important not to set this value too low or too high for the application. If BUFFER_SIZE is very small, the monitor will flush data often and thus affect overall performance. If BUFFER_SIZE is very large, the monitor may compete with the application for node memory resources. The default buffer size is set at 256 kbytes.

4.1.8. **FLUSH_MODE**

The monitor provides two different policies for flushing performance data or trace records from node memory to disk. The policy chosen will determine the action taken when the node's record buffer fills up. If the FLUSH_MODE is non-zero, the default, a node will flush its buffer when the buffer is full. If the value of FLUSH_MODE is 0, the node will not flush its buffer until an explicit flush_trace is called (see instrumentor directives in Section 3.3) or the program terminates. The nodes cease collecting data until the buffer is emptied. The user may, therefore, lose some data if the buffer is full before flush_trace is called.

The default option, emptying buffers whenever they fill up, can cause irregularities in the program's execution, especially if the data buffer is small and flushing is slow. Setting the value of FLUSH_MODE to 0 allows the user to specify when data should be flushed, making the affect on the pattern of execution as small as possible.

4.1.9. PROFILE

A subset of instrumented constructs to monitor may be selected by specifying a profile. The profile is a table of Boolean flags containing an entry for each construct found in the application. Profiles are created by **xinstrument** (see section 3.1.2). The default is to monitor all instrumented constructs. The TRACE_LEVEL parameter takes precedence over any profile setting.

4.1.10. APPL_DB_FILE

The instrumentors encapsulate information about the structure of an application program into an application database file which is subsequently included in the trace file by the monitor. By default, the instrumentors store this information in a file named APPL_DB located in the directory containing the instrumented code. If the name of the application database was changed in **xinstrument** (see section 3.1.1) from the default to another name, that new name must be specified using this parameter.

4.1.11. Examples of .MONITOR Files

To set the values for the above parameters, a file called .MONITOR should be created. Lines in that file should have the form:

```
<parameter name>: <parameter value>
```

Lines beginning with a "#" denote comments. It is important that the last value in the initialization file is followed by a <Return>; otherwise, the monitor will not read the final parameter. The .MONITOR file should be placed in the directory from which the instrumented code is run. The following examples may give a clearer understanding of some of the uses of the .MONITOR file to control the behavior of the monitor.

Example 4-1: If a hostless program is to be run and is expected to generate a lot of data, an initialization file could look like this.

```
TRACE_FILE:        /cfs/mydir/BIG_TRACE
H_TRACE_FILE:      HOST_TRACE
FILE_MODE:         1
FLUSH_MODE:        0
```

The CFS file system is used, so the nodes can flush quickly (since they'll have to flush often). The FLUSH_MODE is set to 0, so the flushes can be indicated in the code. Again, BLOCK_ON_ALL_SYNC is left at1, in case blocking on csends is significant.

Example 4-2: If a program that potentially produces volumes of data is to be used, but perturbation is to be minimized, the size of the trace file is to be limited, and the focus to be only on a few things, then the .MONITOR file should look like that shown below.

```
TRACE_FILE:      /cfs/mydir/FOCUSED_TRACE
H_TRACE_FILE:    HOST_TRACE
FILE_MODE:       1
HOST_PROGRAM:    1
BUFFER_SIZE:     524288
TRACE_LEVEL:     0
```

The buffer size is increased so that only one flush is done, at the end. The TRACE_LEVEL is set to 0; thus, the user can insert instrumentor directives such as insert_marker and begin_block to instrument only those constructs of interest.

4.2. Linking with the Monitor

In order to compile instrumented code, it must be linked with the monitor's libraries. Node programs are linked with the node library nodelib.a, and the host program, if it exists, with the host library hostlib.a. The location of these libraries is specified during installation (see Appendix A). The original source code should also be linked with the appropriate stub file found in the misc directory (either stub.c or stub.f) if it contains instrumentor directives such as insert_marker, begin_block, or flush_trace. For example, a makefile used for compiling instrumented code might contain the following lines:

```
#Specify location of AIMS' monitor
MON_LIB = $(AIMS_DIR)/lib
                 ...
#Link application with monitor libraries
host:
   $(F77) -o host_program $(HOST_OBJS)\
      $(MON_LIB)/hostlib.a $(HOST_LIB)
node:
   $(F77) -o node_program $(NODE_OBJS)\
      $(MON_LIB)/nodelib.a
```

5. Examining the Trace File

The VK and **tally** programs offer different ways of examining the data collected by the monitoring routines. With VK, the trace file can be viewed using a variety of animated views that depict the program's changing state as time passes. **tally** collects and tabulates statistics that reflect the cumulative activity of the program. These two tools are described below, in sections 5.2 and 5.3. Section 5.1 covers the preliminary step of sorting the trace file.

5.1. Sorting the Trace File

Before using the analysis tools, the trace file must be sorted. While the records for each node are already sorted by time within the trace file, the records for different nodes are interleaved, and may therefore be out of order. AIMS provides a `tracesort` tool for sorting trace files.

```
tracesort  <tracefile> > <sorted_tracefile>
```

`tracesort` sorts the events in an AIMS trace file and writes the output to `stdout`.

5.2. The View Kernel

The VK (View Kernel) program animates the trace obtained by executing a parallel program. VK's animated views present information such as: the different constructs that were running; the messages sent between nodes; how long messages waited before being processed; and how long nodes were blocked. Some of the displays scroll along as time passes, showing a segment of the program's history, while others show each state in sequence (drawing over the previous state). Several displays can be viewed at once. The trace file can be stepped through or visualized at high speed, stopping only when certain subroutines are invoked. A source code click-back capability allows easy examination of the source code constructs responsible for trace file events pictured on the display. There are also many ways to customize the displays to better reflect the design of the monitored program.

The displays that VK provides are: an OverVIEW; two "Boxes" views, Circle and Grid; and two communication views, Communication Load and Inbox Sizes.

5.2.1. Invoking VK

```
VK  [-start <start time>]
    [-stop <stop time>]
    [-fixcolors]
    [-help]
    [sorted trace file]
```

VK should be invoked from the uninstrumented source code directory, so that the source files are available for source code click-back. It may be called with up to five arguments, all optional[†]. The application database is used to relate identifiers in the trace file with constructs in the source code. The database that VK uses to view a trace file should therefore be the same as the one used to instrument the program that produced the trace file. To ensure this, the monitor inserts the correct database at the beginning of the trace file, which the VK then reads.

VK's `-start` and `-stop` switches allow the specification of any time segment of the trace file to be viewed. (Pressing the ▶ button after VK pauses continues past the time where the VK stopped.) The values should be non-negative real numbers, which represent the time in milliseconds. If no `-start` flag is present, VK will begin viewing at the beginning of the trace file. If no `-stop` flag is present, VK will view to the end of the trace file (or 1,000,000 msec, whichever comes first).

The overall effect of the `-fixcolors` flag is to disable VK's color editor, which allows the change of colors associated with various constructs. The `-fixcolors` option should be used if VK indicates that it is running out of space for allocating colors. This situation can occur if running several VKs at once, or if viewing a trace file with a very large number of subroutines and blocks. If a message appears indicating that VK cannot allocate a sufficient number of colors appears, VK should be restarted with the `-fixcolors` flag on the command line. If this message appears when VK is started, and others are running, the execution of one of the other VKs should be halted and then restarted as well with the `-fixcolors` flag.

If the `-help` flag is present, VK prints a usage message and exits.

The trace file specified on the command line should be a sorted trace file. If no trace file is specified, the file `inst/TRACE.SORT` is used. VK can view only one trace file at a time.

5.2.2. Using VK

Figure 5-1. VK's Menu

After VK is invoked, a menu like the one in Figure 5-1 appears with a number of buttons. The ▶ button causes VK to start reading and displaying the trace file. If VK was paused in the middle of the file, pressing ▶ will cause it to continue beyond that point. Clicking on ▶ at the end of the trace file has no effect; the ⏮ button will reset VK to the beginning of the trace file. Clicking on ⏸ at any time will pause VK. To step through the trace file one record at a time, use the ⏭ button.

† This X-based application also accepts the `-bg`, `-fg`, `-bd`, `-bw`, `-fn`, and `-xrm` switches. (See Section 6.1.1 for a description of these.)

As shown in Figure 5-2, clicking on **Views** brings up a list of ways to display the trace file. VK's views are: OverVIEW, Boxes (Circle), Boxes (Grid), Communication Load, and Inbox Sizes. OverVIEW is a scrolling view that shows the procedures running on each node, and the messages that are passing between the nodes. The Boxes views display the status of each node at the time of the last trace record. The status information includes the node utilization, the number of pending messages, and the subroutine that is executing. The Communication Load view is a scrolling view that displays the message volume over time. Finally, the Inbox Sizes view displays the volume of pending communication between each pair of nodes.

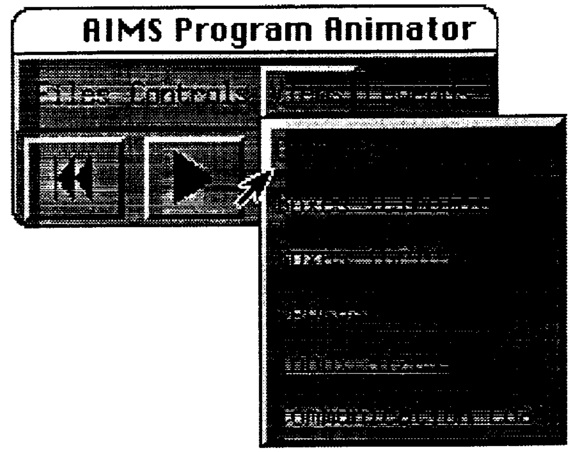

Figure 5-2. Views Menu

Many views may be displayed simultaneously, provided there is only one of each type. To select a view, the **Views** button should be clicked on, showing a list of views, one of which must be selected. To close a view, click on the view's name.

If either the identification of a construct in a view is indeterminable, or a view is displaying something which is not understandable, or the set of various parameters is forgotten, a Help window can be displayed by typing "F1", "h", or "H" in the window. Clicking on the Dismiss button of the Help window will remove it.

The Construct Legend (see Figure 5-3) is displayed when the **Legends** button is clicked. It gives a list of filenames used in producing the trace file. By clicking with the left button on any file name, a construct view window for that file appears. The Node States Legend relates the colors/bitmaps used in the Boxes views with node states. The legends are selected by clicking on the appropriate button in the Legends menu, just as with the Views menu.

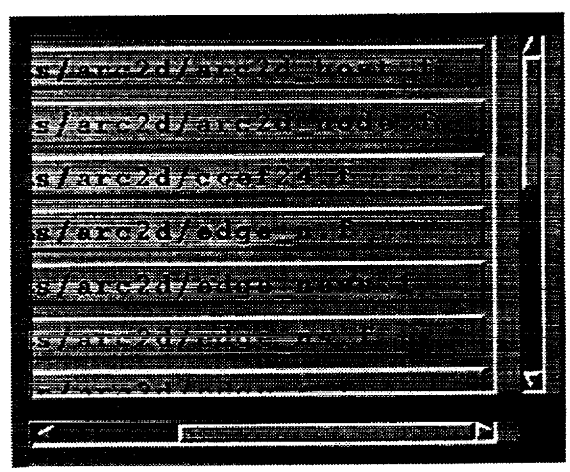

Figure 5-3. Construct Legend

The **By Time** button, under the Controls menu, allows the specification of start and stop times, as on the command line, so VK will pause in the middle of a trace file at a specified time. Times should be specified in milliseconds. To change one of the times, place the cursor over the value and use the backspace key to edit the value. It is important to remember to hit the <Return> key after the changes have been made. Failing to do so will cause the VK not to stop or resume at the specified times.

Clicking on any construct in the Construct View window mentioned above, and then selecting the **Enable Breakpoints** option of the **Controls** menu, causes the VK to stop whenever any node enters one of the selected constructs. (See Figure 5-4.) Note that since the **Constructs View** does not allow the selection of constructs that were not instrumented, VK cannot break at these constructs.

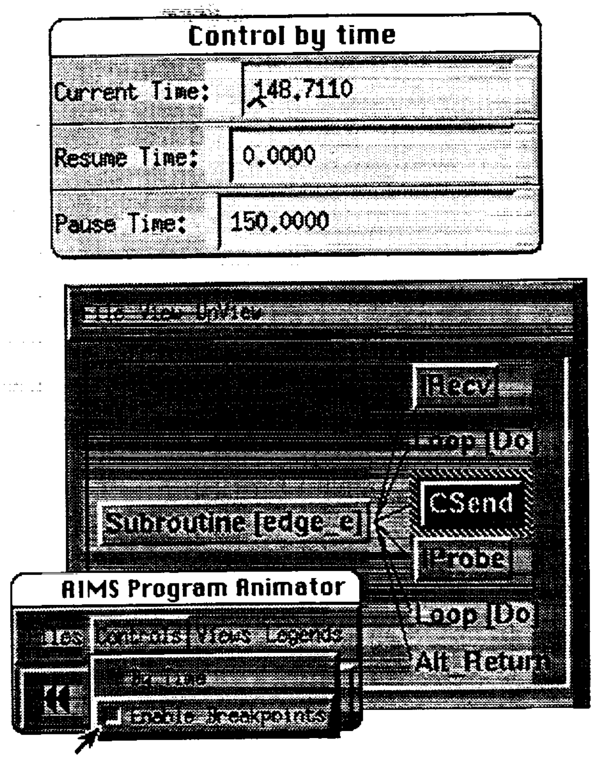

Figure 5-4. Control by Time vs. Break-Points

VK also provides a color editor, Figure 5-5, for modification of colors assigned to the various constructs of an application. For example, it might be useful to color similar routines the same color, or to change colors to heighten contrast. To invoke the color editor, use the ALT-CTRL-right mouse button. A new window will appear to help in the adjustment of the color assignment for that construct. Not all colors may be changed; the color editor will display an error window if (1.) a change is attempted to a fixed color or (2.) a construct does not have a color to change.

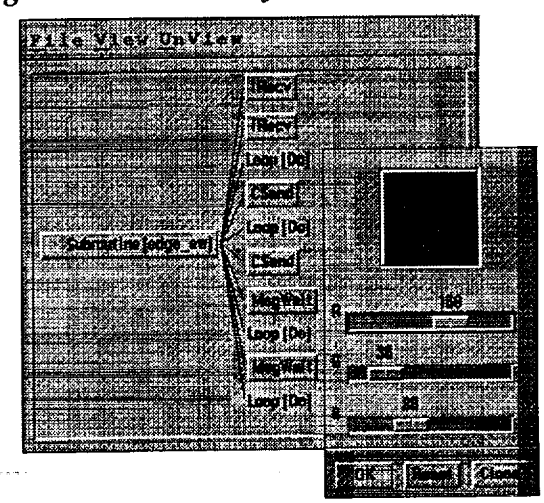

Figure 5-5. Color Editor

The following sections describe each of VK's views in turn. The final section covers some ways to adjust the speed at which VK processes trace records and mouse events.

5.2.3. OverVIEW

The OverVIEW is a scrolling view that displays the procedures being executed by the processors and the messages passing between the processors. Each node is represented by one row of the display, with its number on the left. Figure 5-6, for example, shows a program that ran on sixteen nodes. Constructs are indicated by different colors. Messages are drawn as lines between processors. Periods during which a node is blocked, such as when it is waiting for a message, are shown in the view's background color. The figure shows many instances where a node is blocked, usually waiting for a message. Node 0, for example, is blocked six different times during the period displayed. Any markers the user has set (with `insert_marker` directives) are drawn as thin black lines within procedures, and messages indicating the name of the marker and the time it occurred are printed. A special bitmap is used to show when the nodes flushed their record buffers.

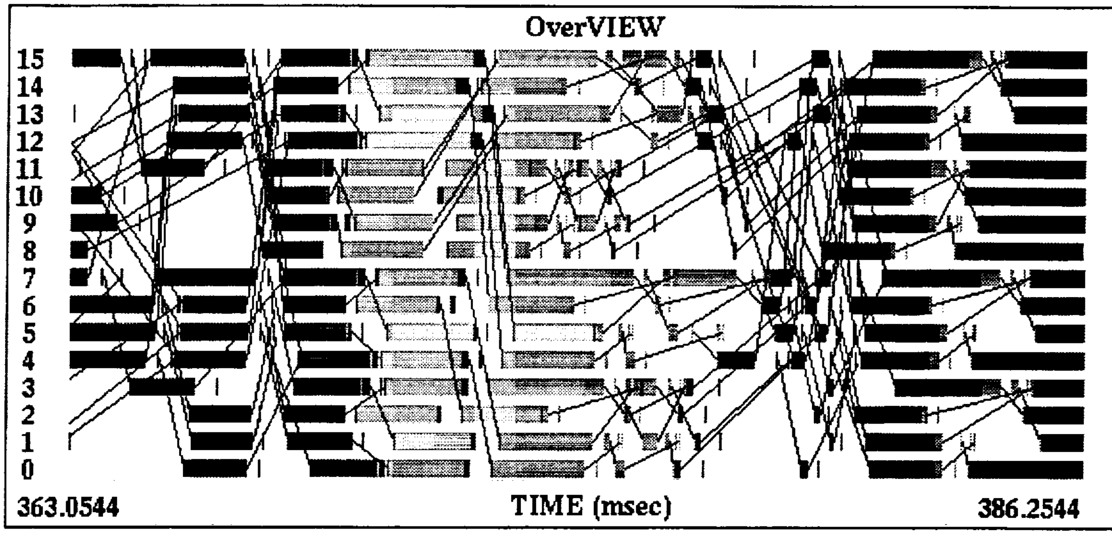

Figure 5-6. OverVIEW

Clicking on a message or procedure bar causes VK to print some information about the relevant construct. If a desired construct is covered by messages, click with the middle button. This causes VK to give information about the procedure, rather than a message. If the `<Shift>` button is held down while clicking the middle button, a window containing the code that initiated the event will be displayed. If a message is clicked on with the left/right button, the following information will be displayed in the shell window used to start up the VK: the time of the send/receive, which node was sending/receiving, the type of the message, and the message size. The code relating to the **sending** of the message will be displayed if the `<Shift>` key is pressed while clicking the left mouse button. The code relating to the **receipt** of the message will be displayed if the `<Shift>` key is pressed while clicking the **right** mouse button.

Key Presses:

Key	Action
b or B	change bar width
d or D	toggle whether dividers are drawn
F1, h, or H	display help window
j or J	change jump factor
m or M	toggle whether messages are drawn
o or O	change node ordering
p or P	print pending messages
r or R	toggle whether markers are drawn
t	change minimum time on x-axis
T	change maximum time on x-axis
x or X	change scale of x-axis

There are many aspects of this view that can be altered. The time scale on the x-axis may be changed as well as the ordering of the nodes on the y-axis. The drawing of messages and markers and small lines between procedures (to make it possible to see recursive procedure invocations) can be made to appear or disappear. The width of the procedure bars can also be changed. The relevant key presses are listed below. Refer to Chapter 6 for more detail on these parameters.

Mouse clicks:

Button	Location	Action
left or right	message line	show information about the message
middle	construct bar	show information about the construct
`<Shift>` middle	construct bar	show construct code
`<Shift>` left	message line	show code that sent the message
`<Shift>` right	message line	show code that received the message

5.2.4. Boxes Views

The Boxes views show the status of each of the processors at the current time. Each processor is indicated by a box; lines or arrows between boxes indicate messages that have been sent but not yet received. As shown in Figure 5-7, each processor box contains five smaller boxes.

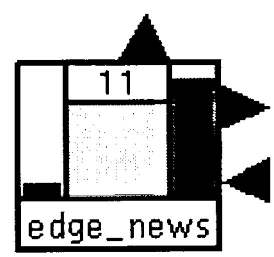

Figure 5-7. A Box

At the top, in the middle, is the number of the processor. Just below that is a rectangle indicating the state of the processor (busy, blocked, or flushing, as indicated by the Node States Legend). At the bottom is the name of the procedure the node is executing. The column to the left indicates the number of pending messages. If the column is full, then there are at least maxInboxCount messages pending. (maxInboxCount is a parameter for the Boxes views, with a default value of 10. Chapter 6 describes how to set values for view parameters.) The column on the right depicts node utilization — the time spent running divided by the total amount of time spent running or blocked. The color and height of the column both reflect the utilization. If node utilization is 90%, a red bar will fill 90% of the column, but if utilization is 10%, a blue bar will fill 10% of the column. (The bar will be black on monochrome displays.)

The Grid version (shown on the left in Figure 5-8) will display information only if a grid topology has been defined in the source code with the `define_grid` and `define_grid_node` user directives (see Section 3.3.6).

Key Presses:

Key	Action
F1, h, or H	display help window

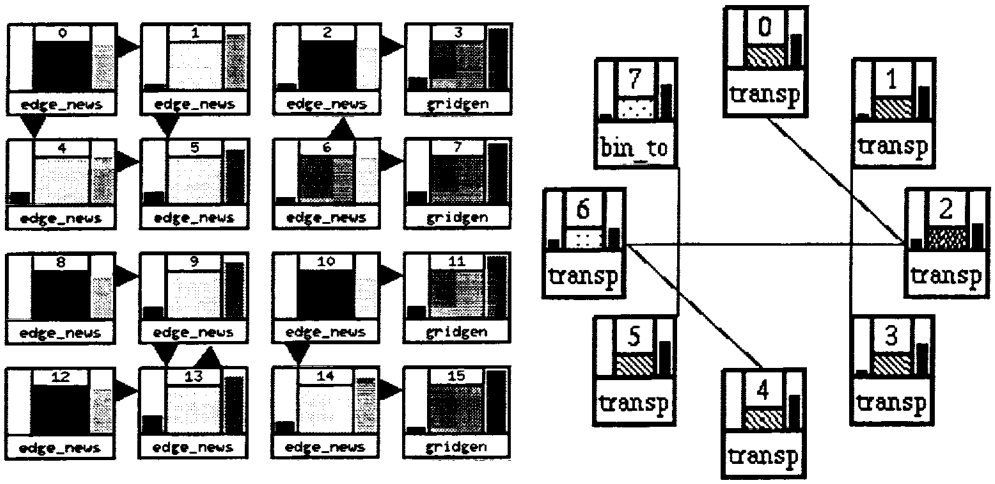

Figure 5-8. Boxes Views: The Grid and Circle Versions

5.2.5. Communication Load

Figure 5-9 shows the Communication Load view, which displays either the cumulative volume (in bytes) or the cumulative pending messages in the system. The view is a scrolling bar chart, with message volume or count on the y-axis and time on the x-axis. If the chart is clicked on, Communication Load prints the volume or count of pending messages at the time clicked. The parameters for rescaling this view are described in Chapter 6.

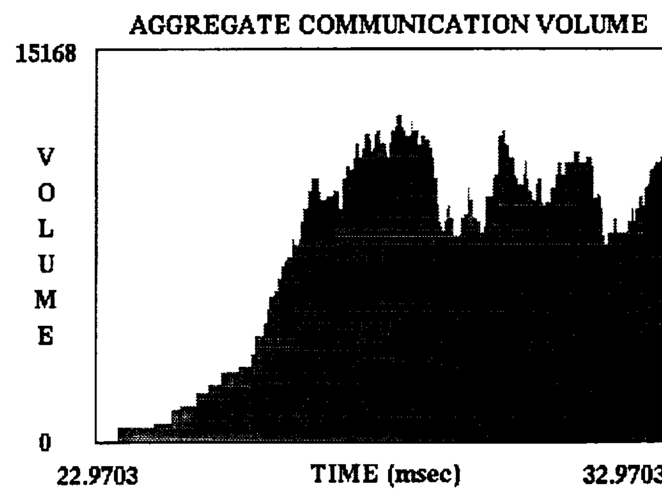

Figure 5-9. Communication Load View

Key Presses:

Key	Action
a or A	change scale-after value
c or C	toggle whether volume or count is drawn
h, H, or ?	display help window
j or J	change jump factor
o or O	change scale-to factor
s/S	change minimum/maximum scaling operator
t/T	change minimum/maximum time on x-axis
x or X / y or Y	change scale of x-axis/y-axis
v	change minimum value on y-axis
V	change maximum value on y-axis
w or W	change "scale-when" factor

5.2.6. Inbox Sizes

The Inbox Sizes view, Figure 5-10, shows the volume of messages, in bytes, pending between each pair of nodes. Colors or bitmaps are used to indicate the volume, as shown by the key on the right side of the view.

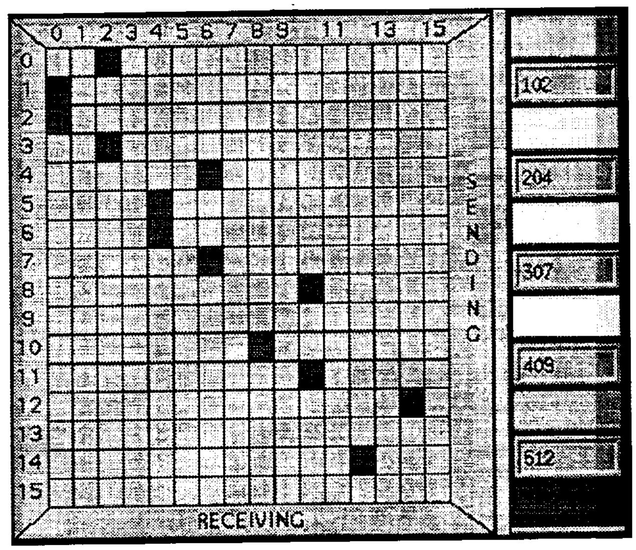

Figure 5-10. Inbox Sizes View

The numbers in the boxes can be changed by putting the mouse in the appropriate box, backspacing to erase the number, typing in the new number, and hitting <Return> when done. In order to change the largest size category and have the others changed automatically, modify the maxSize parameter. The numSizes parameter can be changed in order to modify the number of size categories.

Key Presses:

Key	Action
F1, h, or H	display help window
m or M	change the maximum size
n or N	change the number of size categories

5.2.7. Adjusting VK

If the VK is found to be running too slowly, a non-zero *jump factor* can be specified for the scrolling views (such as *OverVIEW* and *Communication Load*). The value of this factor, a number between 0 and 1, inclusive, specifies a minimum fraction of the window that VK must scroll each time the view needs to scroll (i.e., change the values on its time axis). A value of 0 causes VK to scroll the minimum amount necessary, but a value of 0.5 will ensure that VK always scrolls by half a screen. This means VK will scroll fewer times, and will therefore display things faster, since scrolling is a time-consuming operation. Large jump factors may cause the animation to look somewhat jerky, however. In general, small jump factors of about 0.1 speed up VK greatly without disturbing the display too much.

Another trick to speeding up VK is based on the fact that VK takes longer to draw large views than small ones. So, shrinking some views, especially the scrolling views, may help. Finally, closing views that are not in use will speed up the execution.

If VK is displaying the trace records too quickly, setting pause times via the **By Time** menu, or breaking on certain constructs will slow it down. Decreasing the jump factor will also slow down the animation when viewing scrolling displays. If pictures are moving out of the window too quickly, the scale of the scrolling views can be changed to view a larger segment of the program .

If VK is not responding quickly enough when clicking on a view or menu, the vk.eventsLoop parameter may need to be changed. In order to speed up processing, VK normally checks for X events every eventsLoop iterations (i.e., after processing eventsLoop trace records). The default value for this parameter is 100, but can be set to any non-negative integer. It may turn out that even with a value of 0, VK responds slowly. This is probably because the system running VK maintains a sizable internal queue of X events, all of which must be processed before a mouse event can be processed. If this is the case, specifying a non-zero jump factor can alleviate the problem, as the queue of X events can be processed faster.

5.3. tally

tally generates a list of resource-utilization statistics on node-by-node and routine-by-routine bases.

The routine statistics give information typically provided by profilers with respect to amount of time spent in various functions. In addition, it provides easy access to % communication times in each routine and the significance of the communication time in comparison with the total program execution time. The statistics can help to quickly determine the sections of code which needs to be tuned

The output of tally can be used as input to statistical drawing packages such as Excel and WingZ.

The only input to **tally** is a sorted trace file. **tally** relates identifiers in the trace file with constructs in the source program by using the application database which is part of the trace file information. **tally** places its output consisting of a set of tables, on the standard output as well as in two different summary files.

5.3.1. Calling tally

tally is invoked with a sorted trace file or a `-help` flag as follows:

```
tally [-help]
      [sorted trace file]
```

As with VK (see Section 5.2.1), the database **tally** uses should be the one used to create the trace file. If the `-help` flag is present, **tally** prints a usage message and exits. If no trace file is specified, **tally** uses `inst/TRACE.SORT`. **tally** places its output, a set of tables, on the standard output and into two files: `tally.summary` and `ncpu.summary`.

5.3.2. tally's Output

tally produces several tables of statistics. The first table presents data for each function executing the program. The second table provides communication information per node. Node statistics for each function with communications is also output. The last two tables contain NCPU and routine concurrency statistics [Ref. 4]. The first table is sorted in descending order with respect to function execution times:

1. **Routine**: The routine index and the name of the subroutine.
2. **Busy time**: The amount of time for which the function was performing useful work. This is the amount of time not spent in communication.
3. **Global Blocking**: The amount of time a routine spent in a global blocking operation.
4. **Send Blocking**: The amount of time a routine spent in a send operation.
5. **Receive Blocking**: The amount of time a routine spent in a receive operation.
6. **Life time**: The amount of time taken to execute instructions in this function (excluding the functions called from this function).
7. **Percentage Communication**: This number indicates the percentage of total execution time the routine spent in communication.
8. **Communication Index**: This index takes into account the time spent in the function with respect to the total time spent in the program, as well as the percentage of time spent in communication in this function. The lower this value, the lower the impact on the total program execution time of reducing this function's communication characteristics.

The second table consists of columns that show the aggregate communication characteristics of nodes executing the program. The columns are:

1. **Node number**.
2. **Busy time**: The amount of time the node spent not performing communication related work.
3. **Global Blocking**: Amount of time spent in a global blocking operation.
4. **Send Blocking**: Amount of time spent in a send operation.
5. **Recv Blocking**: Amount of time spent in a receive operation.
6. **Life time**: This is the amount of time the node spent executing the program.
7. **Percentage communication**: This number shows the amount of total execution time of the processor spent in communication.
8. **Link Contention**: This percentage represents the total communication time a node spent in contention.

The next set of tables provides statistics for routines which perform communications. The statistics for each of the routines are presented in the form of independent tables. Each table has entries similar to the ones in the second table. All the tables described above are directed to standard output and stored in the "tally.summary" file. In addition, the NCPU and routine concurrency statistics are computed and directed to the standard output while being stored in a file called `ncpu.summary`.

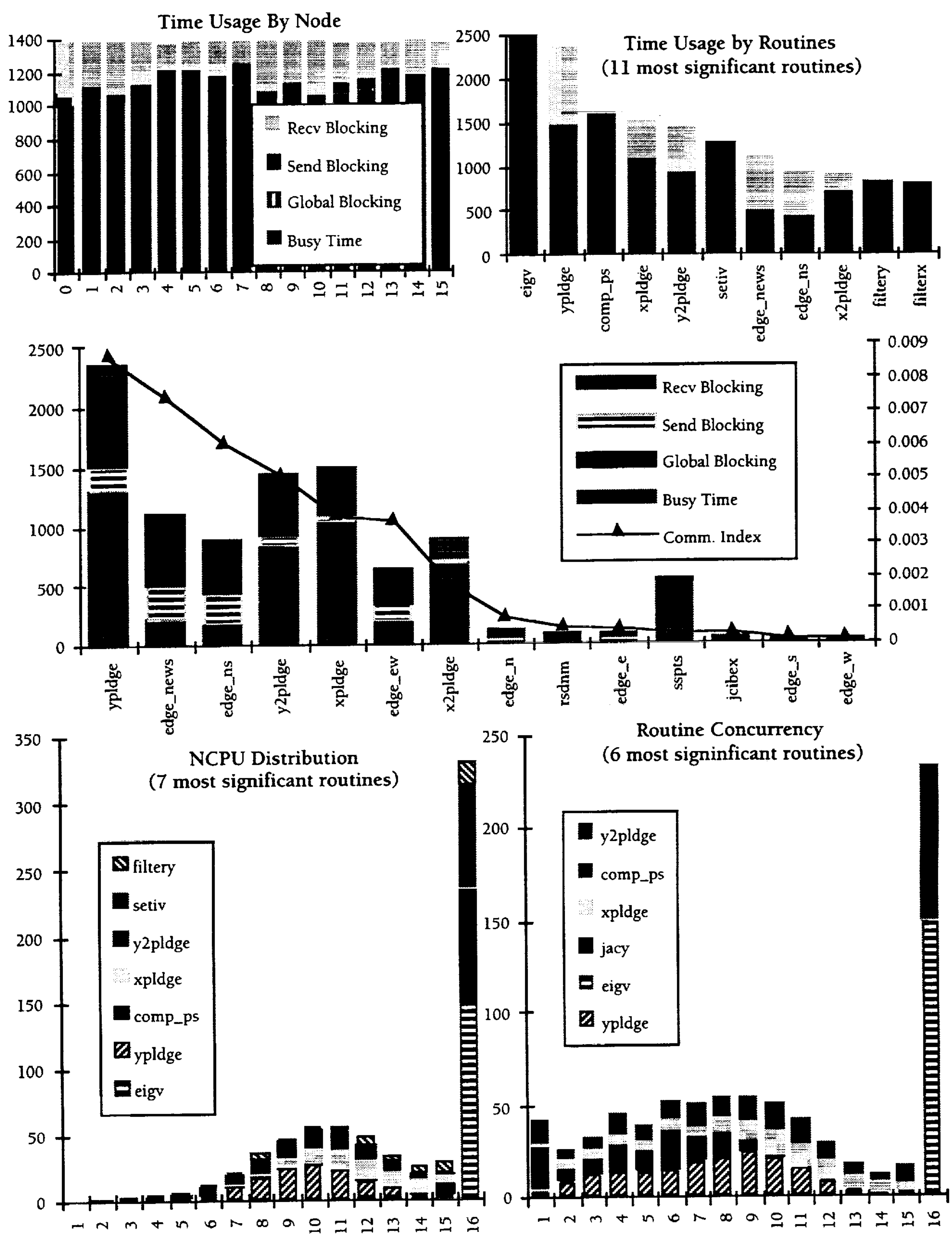

Figure 5-11. A Potpourri of Graphs Created by Excel 4.0 From tally Output

The NCPU for a given subroutine and a given k is the amount of CPU time used by that subroutine when k processors are busy, divided by k. For example, the NCPU data for a particular application is plotted in the lower left-hand-corner of Figure 5-11. It is a highly parallel program: with all (16) of the processors concurrently busy for 325 msecs. During most of that time, subroutine `eigv` is executing. If a subroutine spent much time executing when only a few nodes were busy, this may indicate that the routine inhibits parallelization. In other words, the subroutine may function as a bottleneck.

The Routine Concurrency data for the same trace is plotted in the lower right-hand-corner of Figure 5-11. It indicates the amount of time spent by each subroutine when k copies were executing simultaneously. This view indicates the degree to which each routine was parallelized. If a routine never has more than a few copies running simultaneously, it may indicate that the routine is inherently sequential. Note that this property differs from that of inhibiting parallelism for all subroutines, as described above with the NCPU chart. As expected, `eigv` was the most parallelized routine: it executes concurrently on all the processors for 150 msec.

A great deal of information is output by **tally**. So, we recommend using a statistical drawing package to look at the data. To facilitate that process, each row of tally's tables is a list of numbers or strings separated by tabs. Tables are preceded by a title and separated by a blank line

6. Customizing AIMS

AIMS tools have many parameters that allow one to change things like fonts, initial window sizes, default locations of the trace file and application database, and specific features of VK's views. The parameters have names like `xinstrument.height`, and `vk.overview.font`. The first two sections below explain how to change the default values of the parameters and how to change the values of certain parameters at run-time. The last section contains a list of AIMS' parameters, each of which is fully documented in Appendix D.

6.1. Setting Defaults for Parameters

Defaults for each of AIMS' parameters, which are listed in Section 6.3, are built into the system. However, there are a number of ways one can override these defaults. Indeed, the user may have to if, for example, the default fonts are not available on their system. The defaults can be set on the command-line, or in one of several default files.

6.1.1. Specifying Defaults on the Command Line

Several switches are provided to set values for parameters on the command-line. These are `-bg` and `-fg` for background and foreground, `-bd` and `-bw` for border color and border width, and `-fn` for font. For example, one might type "`VK -bg black -fg chartreuse`" for a glow-in-the-dark look. AIMS' tools allow you to specify a trace file and application database on the command line, as well as specifying other values. (These are described in the sections documenting the use of each tool.) In addition, you can use the `-xrm` switch to specify the value of *any* parameter, by following the switch with the full name of the parameter, a colon, and the parameter's value. For example, "`VK -xrm vk.overview.messageColor:magenta`" would make it very easy to spot the messages that OverVIEW draws.

6.1.2. Specifying Defaults in Files

AIMS looks in several files, including `.Xdefaults`, for defaults. To specify default values in one of these files, add lines to the file of the form "`<default name>:<default value>`". The * notation may be used to specify several defaults with one line. For example, the line "`vk.*.borderWidth:5`" will set the border width of all of VK's views to 5. (The * notation is discussed more fully in many X manuals. See, for example, Section 11.4 in Volume One, the Xlib Programming Manual, by Adrian Nye.) Lines in a default file that begin with an exclamation point are treated as comments. A small default file is shown in Figure 6-1.

6.1.3. How AIMS Finds Defaults

Like many X-based applications, AIMS looks for defaults in four sources in the following order, until it finds a match:

- Command line

- File named in the XENVIRONMENT variable (or .Xdefaults file, if XENVIRONMENT is not set
- Database created by the xrdb program (or .Xdefaults file, if xrdb has not been run)
- /usr/lib/X11/app-defaults/Aims

Thus a command-line value takes precedence over a value in your .Xdefaults file, which in turn takes precedence over one in the system defaults file. If no default value is present in any of the four sources, AIMS uses its built-in defaults (listed in Appendix D with each parameter).

```
! Set default trace file for all X-based tools
*.traceFile: inst/tsort

! Set fonts for VK
vk.*.font: *lucida-medium-r-normal-sans-12-*
vk.help.font: *fixed*medium*-r-*-10-*

! Set jump factor to scroll faster
vk.*.jumpFactor: 0.15

! Position the OverVIEW, and make it long
vk.overview.x: 10
vk.overview.y: 200
vk.overview.width: 800
```

Figure 6-1. An Example of X-defaults

6.2. Changing VK's Parameters Dynamically

VK allows you to change the value of many parameters while the program is running. These *dynamic* parameters are changed by pressing a key in the appropriate window, or by editing in the Preferences window.

The Preferences window allows you to change VK's stop and resume times. To do this, bring up the Preferences window, put the cursor over the value you want to change, and use Emacs-like commands† to edit the value. Remember to hit <Return> in the window when you are done, or the change will not take affect.

To change a parameter with a key press, you simply move the cursor to the appropriate window and press the key corresponding to the parameter you want to change. If necessary, a small window appears where you can enter a new value. The current value is displayed; you can erase it by backspacing or typing <Control>-U. To enter a new value, just hit

† In addition to the usual key strokes and backspace, <Control>-A moves the pointer to the beginning of the line, <Control>-E moves it to the end, <Control>-K deletes everything to the right of the pointer, and so forth.

<Return> or click on the **Okay** button. If you enter a value that is not legal, the terminal beeps. Clicking on sets the value back to the previous value, and clicking **Cancel** resets the value and removes the editing window.

The following are the keys corresponding to the different parameters.

Parameter	Key
vk.<view>.minTime	t
vk.<view>.maxTime	T
vk.<view>.jumpFactor	j or J
vk.<view>.minValue	v
vk.<view>.maxValue	V
vk.<view>.minScalingOp	s
vk.<view>.maxScalingOp	S
vk.overview.showMarks	r or R
vk.overview.showMessages	m or M
vk.overview.barWidthFactor	b or B
vk.overview.drawDividers	d or D
vk.commLoad.volumeOrCount	c or C
vk.commLoad.scaleToFactor	o or O
vk.commLoad.scaleWhenFactor	w or W
vk.commLoad.scaleAfterValue	a or A
vk.inboxSizes.numSizes	n or N
vk.inboxSizes.maxSize	m or M
(node ordering in overview)	o or O

6.3. A Listing of AIMS' Parameters

vk.applicationDatabase
vk.boxes.maxInboxCount
vk.boxes.spectrumSize
vk.boxes.breakpointsEnabled
vk.circle.background
vk.circle.borderColor
vk.circle.borderWidth
vk.circle.font
vk.circle.foreground
vk.circle.height
vk.circle.width
vk.circle.x
vk.circle.y
vk.clickback.big.font
vk.clickback.medium.font
vk.clickback.small.font
vk.commLoad.background
vk.commLoad.borderColor
vk.commLoad.borderWidth
vk.commLoad.countColor
vk.commLoad.font
vk.commLoad.foreground
vk.commLoad.height
vk.commLoad.jumpFactor
vk.commLoad.maxCount
vk.commLoad.maxScalingOp
vk.commLoad.maxTime
vk.commLoad.maxVolume
vk.commLoad.minCount
vk.commLoad.minScalingOp
vk.commLoad.minTime
vk.commLoad.minVolume
vk.commLoad.scaleAfterValue
vk.commLoad.scaleToFactor
vk.commLoad.scaleWhenFactor
vk.commLoad.volumeColor
vk.commLoad.volumeOrCount
vk.commLoad.width
vk.commLoad.x
vk.commLoad.y
vk.eventsLoop
vk.fixColors

vk.grid.background	vk.grid.borderColor	vk.grid.borderWidth
vk.grid.font	vk.grid.foreground	vk.grid.height
vk.grid.width	vk.grid.x	vk.grid.y
vk.help.font	vk.inboxSizes.background	vk.inboxSizes.borderColor
vk.inboxSizes.borderWidth	vk.inboxSizes.font	vk.inboxSizes.foreground
vk.inboxSizes.height	vk.inboxSizes.maxSize	vk.inboxSizes.numSizes
vk.inboxSizes.width	vk.inboxSizes.x	vk.inboxSizes.y
vk.menu.background	vk.menu.borderColor	vk.menu.borderWidth
vk.menu.font	vk.menu.foreground	vk.menu.title.font
vk.menu.title.foreground	vk.nodeState.font	vk.nodeState.notTracing.color
vk.nodeState.running.color	vk.nodeState.blockedSendingColor	vk.blocked.color
vk.nodeState.blockedReceiving.color	vk.nodeState.blockedGlobal.color	vk.overview.background
vk.overview.barWidthFactor	vk.overview.borderColor	vk.overview.borderWidth
vk.overview.font	vk.overview.foreground	vk.overview.height
vk.overview.highlightColor	vk.overview.jumpFactor	vk.overview.maxTime
vk.overview.messageColor	vk.overview.minTime	vk.overview.showDividers
vk.overview.showMarks	vk.overview.showMessages	vk.overview.width
vk.overview.x	vk.overview.y	vk.startTime
vk.stopTime	vk.timeStep	vk.traceFile
vk.traceRecord.background	vk.traceRecord.borderColor	vk.traceRecord.borderWidth
vk.traceRecord.font	vk.traceRecord.foreground	vk.traceRecord.height
vk.traceRecord.width	vk.traceRecord.x	vk.traceRecord.y
vk.utilizationLegend.font		

7. References and Bibliography

[1] T. Lehr, Z. Segall, D. Vrsalovic, E. Caplan, A. Chung & C. Fineman. "Visualizing Performance Debugging". *Computer*, October 1989, pp. 38-51.

[2] G. A. Geist, M. T. Heath, B. W. Peyton, P. H. Worley "PICL — A Portable Instrumented Communication Library" Tech Report ORNL/TM-11130, Oak Ridge National Laboratory. May 1990.

[3] M. Heath and J. Ethridge. "Visualizing the Performance of Parallel Programs". *IEEE Software*, Vol. 8, No. 5, Sept. 1991, pp. 29-39.

[4] T. E. Anderson and E. D. Lazowska. "Quartz: A Tool for Tuning Parallel Program Performance". In *Proceedings of SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, May 1990, pp. 115-125.

- Mehra, P., "Grammar-Driven Interpretation of Tracefiles: Applications in Modeling and Visualization of Message-Passing Parallel Programs" submitted to *The 1994 Scalable High-Performance Computing Conference(SHPCC 94)*, Knoxville, TN May 1994.
- Mehra, P., C. Schulbach, and J. Yan, "A Comparison of Two Model-Based Performance Prediction Techniques for Message Passing Parallel Programs". Submitted to ACM Sigmetrics Conference at Nashville, TN, May 16-20, 1994.
- Mehra, P., M. Gower, and M. Bass, "Automated Modeling of Message-Passing Programs," *Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 94)*, IEEE Computer Society Press, Durham, NC, Jan. 1994.
- Sarukkai, S., "Scalability-Analysis Tools for SPMD Message-Passing Parallel Programs," *Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, IEEE Computer Society Press, Durham, NC, Jan.1994.
- Sarukkai, S., and Gotwals J., "Analyzing Data Structure Movements in Message Passing Programs," Submitted to ACM Sigmetrics Conference at Nashville, TN, May 16-20, 1994.
- Sarukkai, S., and Jerry Yan, "Integration of Perturbation Analysis and Application Monitoring Tools for Message Passing Parallel Programs, " submitted to *IEEE Transactions on Parallel and Distributed Computing Systems.*
- Yan, J. C. and C. E. Fineman, *Modeling Parallel Programs and Multiprocessor Architectures with AXE*, Contractor Rep.177582, NASA Ames Research Center, Moffett Field, CA, May1991.
- Yan, J. C., Schmidt M. and Sarukkai, S., "Monitoring the Performance of Multidisciplinary Applications on the iPSC/860", submitted to *The 1994 Scalable High-Performance Computing Conference(SHPCC 94)*, Knoxville, TN May 1994.
- Yan, J., C. "Performance Tuning with AIMS - An Automated Instrumentation and Monitoring System for Multicomputers," *Proc. 27th Hawaii Int'l. Conf. on Systems Sciences*, ACM, Jan. 1994.
- Yan, J., C. Fineman, P. J. Hontalas, M. Schmidt, S. Listgarten, P. Mehra, S. Sarukkai, and C. Schulbach, *The Automated Instrumentation and Monitoring System (AIMS) Reference Manual*, NASA Ames Research Center, Moffett Field, June 1993.

Appendix A. Installation Guide

Installing AIMS requires three steps: creating the source tree, compiling the system, and installing the executables. Various portions of AIMS may be installed on different machines. The source code is distributed in `tar` format. Once the `tar` file has been obtained, a directory for AIMS should be created and the `tar` file should be "untarred" into that directory. For example,

```
mkdir aims_source
mv aims.tar aims_source
cd aims_source
tar xof aims.tar
rm aims.tar
```

After untarring the file, the source directory should have a `Makefile` and nine subdirectories: `common/`, `example/`,`misc/`, `monitor/`, `poem/`, `poem_spec/`, `parsers/`, `tools/`, and `notes/`. Once the source tree has been created, the source can be compiled. To do this, the `Makefile` in the top-level directory should be edited first. The following may need to be changed: compilation directives, installation directories, location of X libraries, location for default files, and other system-specific definitions. These are discussed in turn below.

The compilation directives indicate which parts of AIMS are to be created. A component will be "made" only if its value is 1. For example:

```
MONITOR=0
INSTRUMENTORS=0
VK=1
TALLY=1
TRACESORT=1
```

When AIMS is "made" and installed in the appropriate directories, the items with a value of 1 are "made" and installed.

The installation directories indicate where the executables should be located. The directory for the monitor is specified separately, as it is a library rather than an executable. For example:

```
INSTALL_DIR = $(HOME)/bin
MONITOR_INSTALL_DIR = $(HOME)/lib
MAN_INSTALL_DIR = $(HOME)/man
```

The monitor installation directory should be remembered when linking the instrumented code with the monitor libraries.

In the next section of the Makefile, the location of the X include files and the libraries should be specified. The values of the DEFAULTS_FILE and DEFAULT_DEST macros are used for installing the system's default. Some default files are located in AIMS' misc/ subdirectory. They are defaults.sgi, defaults.sunos.color, defaults.ultrix.color, and defaults.ultrix.mon. One of these should be specified as the default file, or a new one should be created as described in Chapter 6. The system default file is installed in DEFAULTS_DEST, which is normally /usr/lib/X11/app-defaults/Aims. If DEFAULTS_DEST is /dev/null, no system default file will be installed, and AIMS will use built-in values.

Next comes a variable that defines the target platform the system is being built for. That is, it defines the target machine and source language. The latest version of AIMS supports C and Fortran under the NX operating system on IPSC/860's. CURRENTLY, A DIFFERENT AIMS SYSTEM MUST BE BUILT FOR EACH PLATFORM. THIS MEANS THAT IF THERE ARE USERS OF DIFFERENT PLATFORMS, THERE MUST BE TWO COLLECTIONS OF BINARIES (e.g. one for Fortran on the host, one for C on the node, etc.). The libraries can be shared for a given architecture.

Finally, there are a few other system-specific definitions required for compiling the system. They are grouped into three sections, depending on whether the machine is running IRIX, Sun OS, or Ultrix. Uncomment the lines corresponding to the system of choice through the removal of the pound signs, ensuring the other lines are commented out. When all changes have been made to the Makefile, "make" should be typed from the source directory. The system will take about 30 minutes to compile. When the compilation has completed, "make install" should be typed to install the system in the directories specified in the Makefile. If the full system was created, the following will be in the installation directory:

atopg	translates AIMS tracefiles to ParaGraph format
inst-f77-nx-node inst-f77-nx-host inst-c-nx-node inst-c-nx-host	programs that instrument source code
load-f77-nx-node load-f77-nx-host load-c-nx-node load-c-nx-host	programs used by the instrumentor to load files and create/update an application database
tally	program that tabulates statistics from the trace file
tracesort	program which sorts a trace file
xinstrument	X-based instrumentor
VK	program that graphically displays the trace file

The monitor installation directory will contain two files, nodelib.a and hostlib.a.

Appendix B. From Source Code to Trace Records

The tables on the following pages show the trace records generated by various source code constructs, along with the names of the monitor routines that produce the trace records†. Only those constructs listed in the tables produce trace records. Among the constructs that do *not* produce trace records are the following: `cprobe`, `csendrecv`, `gcol`, `gcolx`, `gsendx`, `hsend`, `hsendrecv`,`isendrecv`, `killproc`, `msgcancel`, `waitall`, and `waitone`.

Many trace records are produced only conditionally; the conditions are listed in the fourth through seventh columns of the tables. The fourth column specifies if the construct must occur on the host or node in order to generate a record. If no value is specified, then the record is produced in either case. Many constructs must be selected in the file's profile before they generate trace records. Such constructs are indicated with a check mark in the fifth column. The value of the monitor parameter EVENT_LEVEL affects the production of many records; this is indicated in the sixth column. For example, a 1 there indicates that the record will be produced only if the event level is at least 1. (If this column has no entry, then the record will be produced regardless of the event level, even if the level is negative.) The last column indicates other miscellaneous conditions that may apply.

Source Code Construct	Monitor Event recorder	Trace Record	Conditions on Production of Trace Record: Node/ Host	Profile	Event Lvl.	Other
Beginning of program	mon_init	MON_BEGIN	Node			Only node 0 sends this.
	start_trace	TRACE_BEGIN	Node		0	
	proc_begin	PROC_BEGIN	Node	√	1	
Call to set_pid	mon_setpid					
Call to load	mon_init					
	start_trace	TRACE_BEGIN	Host		0	
	proc_begin	PROC_BEGIN	Host	√	1	
End of program	stop_trace	TRACE_END			0	
	mon_term	FLUSH_BEGIN	Node			
		FLUSH_END	Node			
		MON_END				

† Event recorders that apparently produce no trace records perform other functions for the monitor.

Source Code Construct	Monitor Event recorder	Trace Record	Conditions on Production of Trace Record: Node/ Host	Profile	Event Lvl.	Other
STOP statement	proc_ends stop_trace mon_term	PROC_END TRACE_END FLUSH_BEGIN FLUSH_END MON_END	 Node Node	√	1 0	
Call to relcube	proc_end stop_trace mon_term	PROC_END TRACE_END MON_END	Host Host Host	√	1 0	
Call to killcube	proc_end stop_trace mon_term	PROC_END TRACE_END MON_END	Host Host Host	√	1 0	
Call to begin_trace	start_trace	TRACE_BEGIN			0	
Call to end_trace	stop_trace	TRACE_END			0	
Beginning of function or subroutine	proc_begin	PROC_BEGIN	Node	√	1	
End of function or subroutine	proc_end	PROC_END	Node	√	1	
Return statement	proc_end	PROC_END		√	1	
Call to begin_block	block_begin	BLOCK_BEGIN		√	0	
Call to end_block	block_end	BLOCK_END		√	0	
Call to insert_marker	point_marker	MARKER		√	0	
Call to a global routine†	global_start global_end	GLOBAL_BEGIN GLOBAL_END		√ √	2 2	
Call to csend	sync_send	SYNC_SEND SYNC_SEND_BLK SYNC_SEND_UNBLK		√ √ √	3 3 3	If BOAS‡ = 0 If BOAS = 1 If BOAS = 1
Invocation of isend	async_send	ASYNC_SEND		√	3	

† This includes calls to the following: `gsync, gdhigh, gdlow, gdprod, gdsum, giand, gihigh, gilow, gior, giprod, gisum, gixor, gland, glhigh, gllow, glor, glprod, glsum, glxor, gshigh, gslow, gsprod, gssum, and gopf.`

‡ BOAS is an abbreviation for the monitor's BLOCK_ON_ALL_SYNC parameter.

Source Code Construct	Monitor Event recorder	Trace Record	Conditions on Production of Trace Record			
			Node/ Host	Profile	Event Lvl.	Other
Call to crecv	sync_recv	SYNC_RECV		√	3	If BOAS = 0 and msg. has arrived.
		SYNC_RECV_BLK		√	3	If BOAS = 0 and msg. hasn't arrived, or BOAS = 1
		SYNC_RECV_UNBLK		√	3	If BOAS = 0 and msg. hasn't arrived, or BOAS = 1.
Invocation of irecv	async_recv					
Call to msgwait	monmwait	ASYNC_SEND_BLK		√	3	If waiting on send.
		ASYNC_SEND_UNBLK		√	3	If waiting on send.
		ASYNC_RECV_BLK		√	3	If waiting on receive.
		ASYNC_RECV_UNBLK		√	3	If waiting on receive.
Call to flush_trace	flushtrace	FLUSH_BEGIN	Node			
		FLUSH_END	Node			
Call to define_grid	set_config	DEFINE_GRID				
Call to define_grid_node	define_node	DEFINE_GRID_NODE				
Any construct that produces records	Any event recorder that produces records	FLUSH_BEGIN	Node			If record buffer is full.
		FLUSH_END	Node			If record buffer is full.

Appendix C. Trace Records

This appendix provides information on AIMS' trace records. The first table lists the records, along with their numerical identifiers, trace level thresholds, and formats. The second section describes the format of each record, and the third the meaning of each record.

C.1. A Listing of AIMS Trace Records

The table to the right lists AIMS' trace records. The numerical identifier for each is shown in the second column. This identifier appears in the trace file, rather than the full name of the record. The third column indicates how large the monitor's TRACE_LEVEL parameter must be in order for the record to be generated. Those records that are produced regardless of the value of the TRACE_LEVEL parameter have no entry in the third column. (The full list of conditions under which each of these records is produced can be found in Appendix B.) The fourth column indicates the format of the record. (S: Short Format; C: Code Block Format; M: Message Format; SM: Short Message Format; F: Flush Format; T: Topology Format) The formats are described in the next section.

Name	ID	Trace Level	Format
TRACE_BEGIN	0	0	S
TRACE_END	1	0	S
PROC_BEGIN	2	1	C
PROC_END	3	1	C
BLOCK_BEGIN	4	0	C
BLOCK_END	5	0	C
MARKER	6	0	C
GLOBAL_BEGIN	7	2	C
GLOBAL_END	8	2	C
SYNC_SEND	9	3	M
SYNC_SEND_BLK	10	3	M
SYNC_SEND_UNBLK	11	3	SM
ASYNC_SEND	12	3	M
ASYNC_SEND_BLK	13	3	SM
ASYNC_SEND_UNBLK	14	3	SM
SYNC_RECV	15	3	M
SYNC_RECV_BLK	16	3	SM
SYNC_RECV_UNBLK	17	3	M
ASYNC_RECV_BLK	18	3	SM
ASYNC_RECV_UNBLK	19	3	M
MON_BEGIN	20		C
MON_END	21		S
FLUSH_BEGIN	22		F
FLUSH_END	23		F
DEFINE_GRID	24		T
DEFINE_GRID_NODE	25		T

C.2. Trace Record Formats

There are six different trace record formats, as described below. Each format consists of several fields, all of which are printed on one line in the trace file and displayed by VK's Trace Record view.

C.2.1. Short Format

The short format consists of four fields:.

- Trace record identifier

- Time of event (seconds)
- Time of event (microseconds)
- Node on which event occurred†

This format is used for the following trace records: TRACE_BEGIN, TRACE_END, and MON_END.

C.2.2. Code Block Format

The code block format consists of the fields in the short format, followed by two additional fields:

- File identifier
- Object identifier

The two identifiers are used with the application database to relate trace file events to source code constructs.

This format is used for the following trace records: PROC_BEGIN, PROC_END, BLOCK_BEGIN, BLOCK_END, MARKER, GLOBAL_BEGIN, GLOBAL_END, and MON_BEGIN.

C.2.3. Message Format

The message format consists of the fields in the short format, followed by six additional fields:

- Other node participating in message
- Type of message
- Size of message
- File identifier
- Object identifier
- Message identifier

The file and object identifiers are used with the application database to relate trace file events to source code constructs. The message identifier is used only for asynchronous transmissions.

This format is used for the following trace records: SYNC_SEND, SYNC_SEND_BLK, ASYNC_SEND, SYNC_RECV, SYNC_RECV_UNBLK, and ASYNC_RECV_UNBLK.

C.2.4. Short Message Format

The short message format consists of the fields in the short format, followed by three additional fields:

- File identifier
- Object identifier
- Message identifier

† The host is represented by the number -32768.

The file and object identifiers are used with the application database to relate trace file events to source code constructs. The message identifier is used only for asynchronous transmissions. This format is used for the following trace records: SYNC_SEND_UNBLK, ASYNC_SEND_BLK, ASYNC_SEND_UNBLK, SYNC_RECV_BLK, and ASYNC_RECV_BLK.

C.2.5. Flush Format

The flush format consists of the fields in the short format, followed by seven additional fields:

- Accumulated flush time (seconds)
- Accumulated flush time (microseconds)
- Flush time (microseconds)
- Bytes
- Flush time (seconds)
- Count
- Total bytes

This format is used for the following trace records: FLUSH_BEGIN and FLUSH_END.

C.2.6. Topology Format

The topology format consists of the fields in the short format, followed by two additional fields:

- Row
- Column

This format is used for the following trace records: DEFINE_GRID and DEFINE_GRID_NODE.

C.3. Trace Record Interpretation

The analysis tools read the trace file to find out when a node began a new code block, when it was blocked, when it sent a message, and when it received a message.

The following trace records indicate that a node is entering (or re-entering) a different code block: PROC_BEGIN, PROC_END, BLOCK_BEGIN, and BLOCK_END.

The following trace records indicate that a node has started blocking: GLOBAL_BEGIN†, SYNC_SEND_BLK, ASYNC_SEND_BLK, SYNC_RECV_BLK, and ASYNC_RECV_BLK.

The following trace records indicate that a node has finished blocking: GLOBAL_END, SYNC_SEND_UNBLK, ASYNC_SEND_UNBLK, SYNC_RECV_UNBLK, & ASYNC_RECV_UNBLK.

The following trace records indicate that a node has sent a message: SYNC_SEND, SYNC_SEND_BLK, and ASYNC_SEND.

The following trace records indicate that a node has received a message: SYNC_RECV, SYNC_RECV_UNBLK, and ASYNC_RECV_UNBLK.

---

† Note that time spent in global operations is considered by the analysis tools to represent time spent blocked. This is because in general any significant time spent in such operations is due to synchronization delays.

Appendix D. AIMS Parameters

This appendix describes all of the user-settable parameters of AIMS' X-based tools. It is divided into several sections, which identify the parameters for **xinstrument**, and VK (general parameters) respectively. Each feature is described by its name, the function it performs, the type of its value (e.g., integer, real, string), the default value, and its update mode (*dynamic* features can be changed during run-time, while *static* features have their values fixed throughout the program's execution). Dynamic features are listed with the key used to change its the value during run-time. Some dynamic features are not changed via a key press, but via the Preferences menu. This is also noted where relevant.

D.1. xinstrument Parameters

xinstrument allows you to specify the main window's dimensions, as well as locations of the application database and output directory.

Name:	**xinstrument**.width and **xinstrument**.height		
Function:	These specify the width and height of **xinstrument**'s main window.		
Type:	integer	Default:	350 and 450
Update mode:	dynamic		
Key:	(None. You can resize the window with the mouse.)		

Name:	**xinstrument**.applicationDatabase		
Function:	This specifies a default location for the application database.		
Type:	string	Default:	appl_db
Update mode:	static (But the value can be over-ridden with a command-line argument.)		

Name:	**xinstrument**.outputDirectory		
Function:	This specifies a default directory for the instrumented files.		
Type:	string	Default:	inst
Update mode:	dynamic (You can change the value by editing **xinstrument**'s window.)		

D.2. VK Parameters

VK has many parameters that you can adjust. Those that pertain particularly to the views are covered in Section D.2.2, while the remainder, dealing with the trace file, fonts, menus, and the like, are listed in Section D.2.1.

D.2.1. General VK Parameters

Note that values specified for these parameters on the command-line will override those present in the defaults file. For example, if you specify in your `.Xdefaults` file that vk.traceFile is trace1, but you invoke the VK by typing "`VK trace2`", then you will view the file trace2, not trace1. This is described more thoroughly in Section 6.3.

Name:	`vk.traceFile`		
Function:	This specifies the trace file that will be viewed.		
Type:	string	Default:	`inst/TRACE.SORT`
Update mode:	static		

Name:	`vk.applicationDatabase`
Function:	This specifies a default location for the application database.
Type:	string — Default: appl_db
Update mode:	static (But the value can be over-ridden with a command-line argument.)

Name:	`vk.startTime`
Function:	The views will not display anything until the startTime (which is measured in milliseconds) has been reached in the trace file.
Type:	non-negative real — Default: 0
Update mode:	dynamic (via Resume Time in the Preferences menu)

Name:	`vk.stopTime`
Function:	VK will pause after the stopTime (which is measured in milliseconds) has been reached in the trace file.
Type:	non-negative real — Default: 1,000,000
Update mode:	dynamic (via the Preferences menu)

Name:	`vk.timeStep`
Function:	This specifies a maximum time interval, in msec, between the times of two consecutively displayed trace records. For example, if the current trace record has time 100 msec, and the next has time 103.5 msec, a timeStep of 1msec will cause 3 "fake" records to be created at times 101, 102, and 103. A timeStep of 0 means that no extra records will be created. A non-zero value causes the views to be updated in a fashion that simulates real time, with fewer discontinuities. However, it slows the processing down accordingly.
Type:	non-negative real number — Default: 0 (no extra trace records are generated)
Update mode:	static

Name:	`vk.breakpointsEnabled`
Function:	This determines whether breakpoints will initially be enabled or disabled.
Type:	boolean — Default: 0
Update mode:	dynamic (via the Preferences menu)

Name:	`vk.fixColors`
Function:	A value of 1 for this parameter indicates that VK should not allow you to dynamically change the colors it associates with the procedures. This is useful for instances when VK runs out of space for allocating new colors. If VK indicates that it cannot allocate enough colors, restart VK with this option set to 1. If VK cannot find enough colors upon start-up, and you have other VKs running, you should restart at least one of those with the colors fixed.
Type:	boolean (0 or 1) — Default: 0
Update mode:	static

Name:	`vk.eventsLoop`
Function:	This parameter indicates how frequently VK should check for X events. It will process `vk.eventsLoop` trace records before checking for and processing incoming X events. If the value of this parameter is large, VK may run faster, since checking for events can be time-consuming (at the same time, VK's response to user input may be slowed).
Type:	positive integer — Default: 100
Update mode:	static

Name:	`vk.clickback.small.font`, `vk.clickback.medium.font`, `vk.clickback.big.font`, `vk.utilizationLegend.font`, `vk.nodeStateLegend.font`, `vk.help.font`, and `vk.menu.font`
Function:	These specify the fonts to be used for the clickback windows, the help windows, the node state legend, the utilization legend, and the menus, respectively. The font for the help menus should be a fixed-width font.

Type:	string (The string value should be a pattern matched by one of the fonts available on your display. You can see the list of available fonts by running the program xlsfonts, and view the fonts individually with the program `xfd`.)
Default:	6x10 Update mode: static

Name:	`vk.menu.background` and `vk.menu.foreground`
Function:	These specify the background and foreground colors for the VK's menus.
Type:	string (This should appear in the file `/usr/lib/X11/rgb.txt`, or the analogous file on your system.)
Default:	background is white, foreground is black Update mode: static

Name:	`vk.menu.borderColor` and `vk.menu.borderWidth`
Function:	These specify the color and width of the menus' borders.
Type:	borderColor is a string and borderWidth is a non-negative integer. (The string value should appear in the file `/usr/lib/X11/rgb.txt` or the analogous file on your system. Upper and lower bounds for the `borderWidth` are determined by the program.)
Default:	`borderColor` is dark turquoise, `borderWidth` is 2
Update mode:	static

Name:	`vk.menu.font`
Function:	This specifies the font to be used for the menus.
Type:	string (The string value should be a pattern matched by one of the fonts available on your display. You can see the list of available fonts by running the program xlsfonts, and view the fonts individually with the program `xfd`.)
Default:	6x10 Update mode: static

Name:	`vk.menu.title.foreground` and `vk.menu.title.font`
Function:	These specify the foreground color and font to be used for the menus' titles.
Type:	string (The foreground value should appear in the file `/usr/lib/X11/rgb.txt` or the analogous file on your system. The font value should be a pattern matched by one of the fonts available on your display. You can see the list of available fonts by running the program `xlsfonts`, and view the fonts individually with the program `xfd`.)
Default:	foreground is black, font is 6x10 Update mode: static

D.2.2. View Parameters

The views that make up AIMS' View Kernel have many features that you can customize. The first section below describes features common to all of the views. The second and third sections describe features common to certain classes of views, the scrolling views and histogram views. Finally, the fourth section describes those features unique to specific views.

D.2.2.1. General View Parameters

The view kernel currently has the following views: OverVIEW, Boxes (Circle/Grid), Communication Load, and Inbox Sizes. Each of these views provides the following options: x coordinate, y coordinate, width, height, background, foreground, border color, border width, and font. The `<view>` portion of the name should be replaced by one of: overview, circle, grid, commLoad, or Inbox Sizes.

Name:	`vk.<view>.x` and `vk.<view>.y`
Function:	The x and y coordinates specify the position of the window's upper-left corner (actually the upper-left corner of the window's border) when it is opened. If the coordinates are both 0, some window managers will let you position the window with the mouse.
Type:	non-negative integer (upper bounds are determined by the program)

Default:	0 and 0
Update mode:	static (Thus, if a window is initially positioned with its upper-left corner at (100,100), it will always reopen at (100,100), even if it is moved before being closed and reopened.)

Name:	`vk.<view>.width` and `vk.<view>.height`
Function:	These specify the width and height of the view.
Type:	non-negative integer (upper and lower bounds are determined by the program)
Default:	

View	Width	Height
OverVIEW	400	220
Boxes (Circle)	360	360
Boxes (Grid)	240	240
Communication Load	400	220
Inbox Sizes	270	240

Update mode:	dynamic
Key:	(None. You can resize the window with the mouse.)

Name:	`vk.<view>.background` and `vk.<view>.foreground`
Function:	These specify the background and foreground colors for the window.
Type:	string (This should appear in the file `/usr/lib/X11/rgb.txt`, or the analogous file on your system.)
Default:	background is white, foreground is black Update mode: static

Name:	`vk.<view>.borderColor` and `vk.<view>.borderWidth`
Function:	These specify the color and width of the window's border.
Type:	borderColor is a string and borderWidth is a non-negative integer. (The string value should appear in the file `/usr/lib/X11/rgb.txt` or the analogous file on your system. Upper and lower bounds for the borderWidth are determined by the program.)
Default:	`borderColor` is dark turquoise, `borderWidth` is 2
Update mode:	static

Name:	`vk.<view>.font`
Function:	This specifies the font that a view will use.
Type:	string (The string value should be a pattern matched by one of the fonts available on your display. You can see the list of available fonts by running the program xlsfonts, and view the fonts individually with the program xfd.)
Default:	6x10 Update mode: static

D.2.2.2. Scrolling View Parameters

There are several parameters that apply only to the scrolling views (currently these are OverVIEW and Communication Load). The extra features supplied for scrolling views are the jump factor and the delimiters of the time axis.

Name:	`vk.<view>.jumpFactor`
Function:	This specifies a minimum fraction of the window that should be scrolled (shifted to the left) when scrolling is necessary. For example, if the minimum and maximum times are 150 and 200, and a record comes in at 203, the window would only shift left by 3 msec, with a jump factor of zero. However, a jump factor of .2 would cause it to shift by 10 msec (.2 * (200-150)). Higher jump factors cause the window to display trace records faster, since it doesn't have to scroll as many times. However, the scrolling may appear somewhat jerky since the window scrolls in larger amounts.
Type:	real number between 0 and 1, inclusive Default: 0.0
Update mode:	dynamic Key: j or J

Name:	`vk.<view>.minTime` and `vk.<view>.maxTime`
Function:	These specify the minimum and maximum times (in milliseconds) to use for the view's x-axis.
Type:	non-negative real number
Default:	`minTime` is 0.0, `maxTime` is 10.0
Update mode:	dynamic
Key:	`t` for `minTime`, `T` for `maxTime`

D.2.2.3. Histogram View Parameters

There are a number of parameters specific to a certain class of scrolling views, namely the histogram views (currently only Communication Load fits into this category). They are the following: maximum value, minimum value, maximum scaling operator, minimum scaling operator, scale to factor, scale when factor, and scale after value.

Name:	`vk.<view>.minValue` and `vk.<view>.maxValue`
Function:	A histogram view is a scrolling view that depicts the changing value of a certain quantity as time passes. The minimum/maximum values are the initial settings for the y-axis of these views.
Type:	The type varies. For the Communication Load view, these values are integers between 0 and 9,999,999, inclusive.
Default:	see table below

Default:

Name	Default Value
commLoad.minVolume	0
commLoad.minCount	0
commLoad.maxVolume	10000
commLoad.minVolume	100

Update mode: dynamic Key: `v` for `minValue`, `V` for `maxValue`

Name:	`vk.<view>.minScalingOp` and `vk.<view>.maxScalingOp`
Function:	A scaling operator determines whether a min/max value will be adjusted automatically by the program. If the scaling operator is "fixed", the value will change only when the user changes it. If it is "variable", the value will be adjusted automatically by the program. (The way in which the program makes its adjustments is determined by the values of the remaining three parameters described below.)
Type:	string (one of "fixed" or "variable")
Default:	`minScalingOp` is fixed, `maxScalingOp` is variable
Update mode:	dynamic
Key:	`s` for `minScalingOp`, `S` for `maxScalingOp`

The three features that control the automatic scaling are described below, but first we'll give a brief description of how the scaling is done. Automatic scaling on the maximum value works as follows. (It works symmetrically on the minimum value.) If a value needs to be drawn that is greater than the maximum value, the maximum value is increased. If all of the values that are depicted are much smaller than the maximum value, and they have been so for a while, the maximum value is decreased. The value that the maximum value should change to is affected by the scale-to factor. The definition of "much smaller", as used above, is defined by the value of the scale-when factor. And finally, the scale-after value specifies the amount of time during which all of the values should be small before the maximum value is changed.

Name:	`vk.<view>.scaleToFactor`
Function:	If a value must change, this determines the number that it will change to. More specifically, if the maximum value must be changed, it is changed to the new maximum times the

	scaleToFactor. If the minimum value must be changed, it is changed to the new minimum divided by the scaleToFactor.		
Type:	real number between 1 and 5, inclusive	Default:	1.5
Update mode:	dynamic	Key:	o or O

Name:	vk.<view>.scaleWhenFactor		
Function:	This determines when the resolution along the y-axis can be increased. If all of the bars displayed are very short (less than the scaleWhenFactor of chart's height), then the maximum value can be decreased. Similarly, if all of the values are very tall (greater than (1 – scaleWhenFactor) of the chart's height), then the minimum value can be increased.		
Type:	real number between 0 and 1, inclusive	Default:	0.5
Update mode:	dynamic	Key:	w or W

Name:	vk.<view>.scaleAfterValue		
Function:	This determines how long all values must remain small/large (as specified by the scaleWhenFactor) before the max/min value is adjusted. (The number actually represents the number of trace records that must be processed by the view.)		
Type:	integer between 0 and 10,000, inclusive	Default:	100
Update mode:	dynamic	Key:	a or A

D.2.2.4. Specific View Parameters

Some features are specific only to one view. These are described below on a view-by-view basis. **OverVIEW features —**

Name:	vk.overview.showMarks		
Function:	This determines whether or not the view will indicate marks set by the insert_marker directive.		
Type:	boolean (0 or 1)	Default:	1 (marks are indicated)
Update mode:	dynamic	Key:	r or R

Name:	vk.overview.showMessages		
Function:	This determines whether or not the view will draw message lines.		
Type:	boolean (0 or 1)	Default:	1 (messages are drawn)
Update mode:	dynamic	Key:	m or M

Name:	vk.overview.messageColor		
Function:	The messages will be drawn in this color.		
Type:	string (This should appear in the file /usr/lib/X11/rgb.txt, or the analogous file on your system.)		
Default:	blue	Update mode:	static

Name:	vk.overview.showDividers		
Function:	This determines whether small black lines are drawn to separate procedures. The lines help to show invocations of recursive procedures, which normally would not be shown, but the dividing lines can cover up procedures that run for very short amounts of time.		
Type:	boolean (0 or 1)	Default:	0 (dividers are not drawn)
Update mode:	dynamic	Key:	d or D

Name:	vk.overview.barWidthFactor		
Function:	This determines the width of the procedure bars displayed on the view. A value of 1 causes them to be as wide as possible, whereas a value of 0 causes them not to be drawn.		
Type:	real number between 0 and 1, inclusive	Default:	0.7
Update mode:	dynamic	Key:	b or B

Name:	vk.overview.highlightColor
Function:	A highlighted message (one that is clicked on) is drawn in this color.

Type:	string (This should appear in the file `/usr/lib/X11/rgb.txt`, or the analogous file on your system.)		
Default:	deep pink	Update mode:	static

Name:	(None. The node ordering cannot be specified by a named default.)		
Function:	This determines the order in which the nodes are listed on the y-axis.		
Type:	Each node is specified by a non-negative integer between 0 and the number of nodes - 1, inclusive.		
Default:	increasing order from the bottom of the chart to the top		
Update mode:	dynamic	Key:	o or O

Boxes features —

Name:	`vk.boxes.maxInboxCount`		
Function:	This is the maximum inbox count that can be displayed in the view's boxes. (Any higher count is displayed at the maximum height.)		
Type:	integer between 0 and 4096, inclusive		
Default:	10	Update mode:	static

Name:	`vk.boxes.spectrumSize`		
Function:	This is the number of colors used to display the node utilization in the view's boxes.		
Type:	integer between 0 and 100, inclusive		
Default:	16	Update mode:	static

Communication Load features —

Name:	`vk.commLoad.volumeOrCount`		
Function:	This determines whether total message volume or the total number of messages is displayed.		
Type:	string (volume or count)	Default:	volume
Update mode:	dynamic	Key:	c or C

Name:	`vk.commLoad.volumeColor` and `vk.commLoad.countColor`
Function:	This determines the color in which the volume/count are to be displayed.
Type:	string (This should appear in the file /usr/lib/X11/rgb.txt, or the analogous file on your system.)
Default:	`volumeColor` is orchid, `countColor` is cornflower blue
Update mode:	static

Inbox Sizes features include numSizes and maxSize.

Name:	`vk.inboxSizes.numSizes`		
Function:	This determines the number of size categories that will be displayed. (This is one less than the number of colors that the view will use to depict the volume of pending messages.)		
Type:	integer between 1 and 16, inclusive	Default:	5
Update mode:	dynamic	Key:	n or N

Name:	`vk.inboxSizes.maxSize`		
Function:	This determines the maximum size of the size categories. (This is the number in the lowest dialogue box on the right side of the view.)		
Type:	integer between 1 and 99,999, inclusive	Default:	512
Update mode:	dynamic	Key:	m or M

Appendix E. Converting AIMS Trace Files for ParaGraph

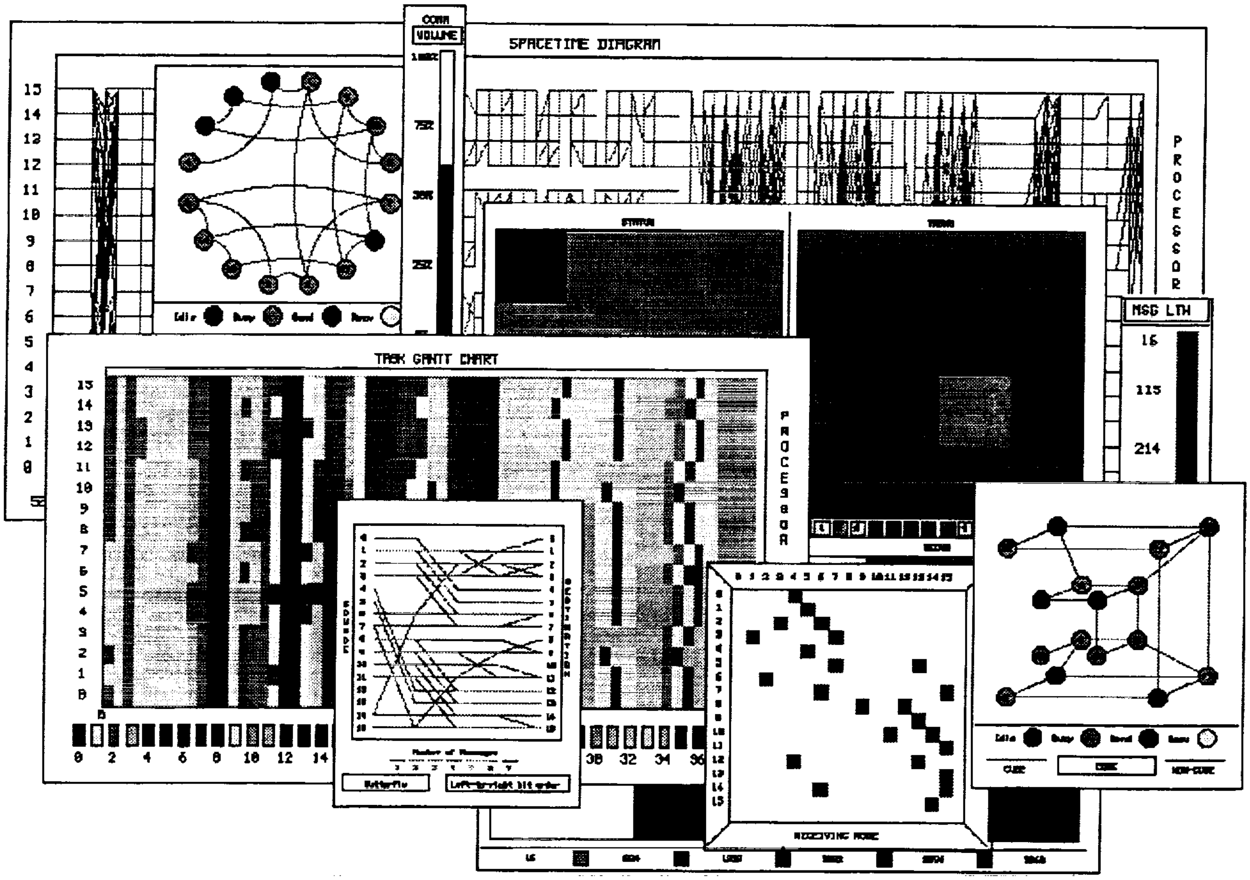

Figure E-1. A Potpourri of ParaGraph Views from a Converted AIMS Trace File

Along with AIMS, we also provide a resource called **atopg**. This program allows the user to convert an AIMS trace file into a format readable by ParaGraph. On running an AIMS trace file through **atopg**, views such as the ones displayed above can be produced. The usage of atopg is as follows:

```
atopg aims-trace > pgph-trace
```

The trace file produced by **atopg** can generate the following paragraph views: *Utilization Count*, *Gantt*, *Summary*, *Meter*, and *Profile*; *Communication Traffic (Volume)*, *Spacetime*, *Queues*, *Matrix*, *Meter (Volume)*, *Animation*, *Hypercube*, *Network*, *Node Data*, and *Color Code*; *Task Count*, *Gantt*, *Status*, and *Summary*; *Clock*, *Trace*, *Statistics*, and *Processor Status*. The *Utilization Kiviat*; *Communication Traffic (Count)* and *Meter (Count)*; *Critical Path*, *Phase*, and *Coord Info.* views do not work because of incompatibilities in the AIMS monitor and PICL monitor.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1993	Technical Memorandum

4. TITLE AND SUBTITLE
The Automated Instrumentation and Monitoring System (AIMS) Reference Manual

5. FUNDING NUMBERS
509-10-33

6. AUTHOR(S)
Jerry Yan,* Philip Hontalas, Sherry Listgarten,* et al.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ames Research Center
Moffett Field, CA 94035-1000

8. PERFORMING ORGANIZATION REPORT NUMBER
A-94012

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM-108795

11. SUPPLEMENTARY NOTES
Point of Contact: Jerry Yan, Ames Research Center, MS 269-3, Moffett Field, CA 94035-1000
(415) 604-4381
*Recom Technologies, San Jose, California

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category – 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Whether a researcher is designing the "next parallel programming paradigm," another "scalable multiprocessor" or investigating resource allocation algorithms for multiprocessors, a facility that enables parallel program execution to be captured and displayed is invaluable. Careful analysis of execution traces can help computer designers and software architects to uncover system behavior and to take advantage of specific application characteristics and hardware features. A software tool kit that facilitates performance evaluation of parallel applications on multiprocessors is described in this paper. The Automated Instrumentation and Monitoring System (AIMS) has four major software components: a source code instrumentor which automatically inserts active event recorders into the program's source code before compilation; a run-time performance-monitoring library, which collects performance data; a trace file animation and analysis tool kit which reconstructs program execution from the trace file; and a trace post-processor which compensate for data collection overhead. Besides being used as prototype for developing new techniques for instrumenting, monitoring, and visualizing parallel program execution, AIMS is also being incorporated into the run-time environments of various hardware testbeds to evaluate their impact on user productivity. Currently, AIMS instrumentors accept FORTRAN and C parallel programs written for Intel's NX operating system on the iPSC family of multicomputers. A run-time performance-monitoring library for the iPSC/860 is included in this release. We plan to release monitors for other platforms (such as PVM and TMC's CM-5) in the near future. Performance data collected can be graphically displayed on workstations (e.g. Sun Sparc and SGI) supporting X-Windows (in particular, X11R5, Motif 1.1.3).

14. SUBJECT TERMS
Performan evaluation, Parallel processing, Performance monitoring

15. NUMBER OF PAGES
70

16. PRICE CODE
A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		